P516249.PDF [Page: 3 of 101]

DREO-CR-2001-066

D-SAFIRE Software Design Document

Authors:
Craig Williams, Michael Seymour
Todd Horn

June 2001

Terms of release
The information contained herein is proprietary to Her Majesty and is provided to the recipient on the understanding that it will be used for information and evaluation purposes only. Any commercial use including use for manufacture is prohibited. Release to third parties of this publication or information contained herein is prohibited without the prior written consent of Defence Research and Development Canada.

DEFENCE R&D DÉFENSE
Defence Research Establishment Ottawa

Vantage Point INTERNATIONAL

Vantage Point International Inc
1704 Carling Avenue 2nd floor
Ottawa, Ontario, K2A 1C7
Tel: 613-798-2244 Fax: 613-798-5369
www.vantpoint.com

DREO CR 2001-066

P516249.PDF [Page: 4 of 101]

105-03438/

©HER MAJESTY THE QUEEN AS REPRESENTED BY THE MINISTER OF NATIONAL DEFENCE, 2001

P516249.PDF [Page: 5 of 101]

D-SAFIRE Software Design Document

Authors:
Craig Williams (VPI), Todd Horn (VPI)
Michael Seymour (VPI)

Vantage Point International Inc
1704 Carling Avenue 2nd floor
Ottawa, Ontario, K2A 1C7

Contract W7714-0-0366

Prepared for: Gary Geling
Aerospace Radar and Navigation
Defence Research Establishment Ottawa, Shirley's Bay
3701 Carling Ave, Ottawa, Ontario, K1A 0Z4

June 2001

Terms of release

The information contained herein is proprietary to Her Majesty and is provided to the recipient on the understanding that it will be used for information and evaluation purposes only. Any commercial use including use for manufacture is prohibited. Release to third parties of this publication or information contained herein is prohibited without the prior written consent of Defence Research and Development Canada.

Disclaimer

The scientific or technical validity of this Contract Report is entirely the responsibility of the contractor and the contents do not necessarily have the approval or endorsement of Defence Research and Development.

PS16249.PDF [Page: 6 of 101]

EXECUTIVE SUMMARY

This document describes the software design for D-SAFIRE as part of contract W7714-0-0366. The purpose of the project is to migrate the existing high fidelity radar simulation program, SAFIRE, to an HLA compliant program, D-SAFIRE, which can then operate as part of a distributed simulation. D-SAFIRE will then be used as part of an Air Crewstation Demonstrator for the CF-18.

This document begins with an examination of the requirements for the project. This section breaks down the requirements into four areas. These areas are the requirements for the User Interface, RTI communications, Modifications to SAFIRE and Testing. As part of the requirements three conceptual models are presented as an aid to understanding the components of the required system and what is outside the scope of the system. The Aircraft conceptual model describes how the aircraft components and attributes are related. The conceptual model of the RTI communication illustrates the communication between the distributed components. The conceptual model of SAFIRE and D-SAFIRE shows the delineation between the software components of programs. The final part of the requirements section is a description of some Use Cases. These Use Cases are meant to illustrate the various circumstances and events that will occur for D-SAFIRE.

The software design section analyses each of the requirements and provides solutions to meet them. The solutions are presented as either belonging in the new RTI communication modules or as modifications to the existing SAFIRE code or both. Each of the interactions in the Federation Object Model are analysed and solutions proposed.

The final section describes the methods and software requirements for testing D-SAFIRE. This includes some test flight scenarios which will verify the operation of the distributed radar modelling software. There is also a proposed schedule of tasks for integrating and testing D-SAFIRE.

P516249.PDF [Page: 7 of 101]

TABLE OF CONTENTS

1. Requirements Definition 1
1.1 Introduction 1
1.1.1 Goals for the System 1
1.1.2 Scope 2
1.1.3 Definitions 2
1.2 Requirements 3
1.2.1 User Interface Requirements 4
1.2.2 SAFIRE Modification Requirements 5
1.2.3 RTI Communication Requirements 6
1.2.4 Testing Requirements 6
1.3 Conceptual Models 7
1.3.1 D-SAFIRE Model 7
1.3.2 Aircraft Model 7
1.3.3 RTI Communication Model 8
1.4 Operational Requirements 12
1.4.1 Actors 12
1.4.2 Use Cases 12

2. Software Design 15
2.1 Flow of Execution 15
2.2 Radar Create Request and Radar Destroy Request 20
2.2.1 D-SAFIRE Parent 21
2.2.2 D-SAFIRE Parent's Federate Ambassador 22
2.3 User Interface 23
2.3.1 Disabling Local Control 23
2.4 Radar Object 27
2.4.1 Update Target Processing List 27
2.4.2 Radar Parameters 30
2.4.2.1 Class Descriptions 32
2.4.2.2 Radar_Parameter_Change.radar_mode_ind 33
2.4.2.3 Radar_Parameter_Change.radar_numbar 34
2.4.2.4 Radar_Parameter_Change.radar_az_center 34
2.4.2.5 Radar_Parameter_Change.radar_el_center 35
2.4.2.6 Radar_Parameter_Change.radar_del_az 36
2.4.2.7 Radar_Parameter_Change.radar_acq_ind 37
2.4.2.8 Radar_Parameter_Change.TAGC 37
2.4.2.9 Radar_Parameter_Change.radar_acq_bump 37

P516249.PDF [Page: 6 of 101]

2.4.2.10 Radar_Parameter_Change.radar_acq_az 37
2.4.2.11 Radar_Parameter_Change.radar_acq_vel 37
2.4.2.12 Radar_Parameter_Change.radar_acq_range 37
2.4.2.13 Radar_Parameter_Change.radar_acq_scale 37
2.4.2.14 Radar_Parameter_Change.radar_az 37
2.4.2.15 Radar_Parameter_Change.radar_el 38
2.4.3 Radar Status Report 38

2.4.3.1 Class Definitions 40
2.4.4 Radar Pause Request and Radar Continue Request 41
2.4.5 Update Aircraft 42
2.4.5.1 Class Definitions 44
2.4.6 Time Management 45
2.4.7 Radar_Object Class Definition 47
2.5 D-SAFIRE Federate Ambassador 47

3. Testing 49
3.1 Testing Interface 49
3.2 Test Observer 49
3.3 Test Stub design 51
3.4 Test Plan 52
3.5 Test Flight Scenarios 54
3.5.1 T-Flight Parameters 54
3.5.2 Head-On Flight Parameters 56

References 59

Appendix A - Class Definition of D_SAFIRE_Parent 60

Appendix B - Mode Changes in SAFIRE 63

Appendix C - Additional Class and Structure Definitions 64

Appendix D - Class Definition of the Radar_Object 80

Appendix E - Radar_Object::Run() 86

List of Symbols, Abbreviations, Acronyms and Initialisms 90

P516249.PDF [Page: 9 of 101]

LIST OF FIGURES

Figure 1 Model of D-SAFIRE components 9
Figure 2 Conceptual Models of an Aircraft. 10
Figure 3 Conceptual Model of RTI Communication 11
Figure 4 Control flow Diagram 16
Figure 5 T-Flight initial Conditions 55
Figure 6 Head-On Flight Side View 58

LIST OF TABLES

Table 1 Target Data 29
Table 2 Radar Parameters 32
Table 3 Radar Detection Attributes 39
Table 4 Types of Radar_Status_Reports 39
Table 5 Aircraft Update Parameters 43
Table 6 Attrubutes of the SAFIRE_TSPI_s 43
Table 7 Attributes of an ALLVector 43
Table 8 Attributes of a NEDVector 44
Table 9 Enumeration of the Weapon Configurations 44
Table 10 Observer Command Line Parameters 50
Table 11 T-Flight Parameters 55
Table 12 Head-On Flight Parameters 57

P516249.PDF [Page: 10 of 101]

1. Requirements Definition

1.1 Introduction

D-SAFIRE is part of a larger project to build an Air Crewstation Demonstrator (ACD) capable of simulating the environment of a CF-18 cockpit for human factor studies. One of the goals for the task is to provide a high fidelity radar simulator for the ACD. Since the ACD will operate in a distributed environment a standard interface specification will be used to communicate between components.

Detailed radar modelling software (SAFIRE) has been developed by the Aerospace Radar and Navigation Section of the Defence Research Establishment Ottawa. SAFIRE is capable of realistically simulating most air-to-air modes of an airborne intercept radar. SAFIRE must be modified to operate with the distributed simulation. The modifications will allow SAFIRE to communicate using the High Level Architecture protocols, in order to obtain information about the aircraft it is modelling the radar for as well as the flight dynamics of other aircraft. SAFIRE will also respond to cockpit operator actions and report back realistic radar parameters and detection information.

The High Level Architecture (HLA) is a standard framework that supports simulations with distributed components. HLA was developed by the United States Defence Modelling and Simulation Office (DMSO) of the Department of Defence. The HLA Interface Specification defines a standard for a Run-Time Infrastructure (RTI). An RTI simplifies communication between components of a simulation distributed across multiple platforms. The HLA RTI 1.3 has received status as an IEEE standard (IEEE 1516, 1516.1, 1516.2). The implementation of the RTI interface specification that will be used is HLA RTI-NG 1.3 version 3.1. (The 1.3 is the version of the HLA specification and the 3.1 is the version of the implementation.)

1.1.1 Goals for the System

The goal for this project is to migrate the existing SAFIRE program to a High Level Architecture (HLA) - compliant application, D-SAFIRE. The D-SAFIRE program will then be able to take part in distributed simulations. This will allow the program to interact with the Air Crew Station Demonstrator (ACD) simulator through a RunTime Infrastructure (RTI) interface.

We expect to demonstrate a limited radar search mode capability integrated with the ACD by the end of March, 2001.

P516249.PDF [Page: 11 of 101]

1.1.2 Scope

The scope of call-up #3 on contract W7714-0-0366 (D-SAFIRE) is to complete the design and implement the changes necessary to the current SAFIRE program as well as the RTI communication modules. This document describes how D-SAFIRE is designed and implemented. Some prototyping and testing was done to verify critical aspects of the design and implementation of D-SAFIRE. The D-SAFIRE program will only be required to model one radar mode Range While Search, (RWS) and a maximum of two targets initially. There could be any number of Aircraft involved in the simulation, but only two targets considered for radar detection. This could be expanded in the future to include multiple modes and more targets.

1.1.3 Definitions

Some general terms which are used to describe the structure and functional requirements of the software.

Aircraft	Any airborn vehicle. An aircraft is characterised by a position, velocity, acceleration, angular orientation, angular velocity, radar cross section, electronic counter measures etc.
Ownship	The aircraft that owns the radar sensor.
Radar Sensor	The device on an Aircraft that transmits the radar signal, processes the return signal, and delivers relevant information to the Display. In this case the Radar will be a model of an APG-65 multimode pulse doppler radar.
Display	The device that displays the information returned by the radar sensor of an aircraft.
Target	A target is any aircraft that is eligible to be detected by the radar sensor of another aircraft.
Hit	Any target detected by the radar sensor.
Flight Dynamics Simulator	The entity that generates information about each target's state throughout the simulation
Radar Simulation Operator	The individual who operates the radar simulation software. This person is responsible for starting the software and ensuring correct operation

P516249.PDF [Page: 12 of 101]

Airspace State	An airspace state encapsulates the state of all aircraft in the simulation at a given moment. The airspace state consists of the position, velocity, acceleration and ECM, etc. of all Aircraft at a given moment in the simulation. The airspace state also consists of the current radar mode of operation.
Target Processing List	A list of all targets and their current state.
Radar Status Report	A list of all hits detected by the radar sensor.
High Level Architecture (HLA)	A standard framework that supports simulations distributed components.
Federate	An individual component of a simulation. A federate accomplishes a specific task within a federation.
Federation	A collection of federates that work together in a simulation
RunTime Infrastructure	A standard interface defined by the HLA Interface Specification for communication between HLA comliant components.
Rtiexec	A process which controls the overall federation operation.

1.2 Requirements

The requirements for D-SAFIRE are examined in this section. This section provides a more specific description of what has to be completed. The requirements can be broken down into four specific areas. One, the User Interface requirements describe what is necessary for interaction between the DSAFIRE program and the Radar Operator. Two, the SAFIRE modification requirements describe the modifications to the existing SAFIRE program that are required. These requirements describe the modification that will allow the RTI communication modules to input and extract data from SAFIRE. Three, the RTI Communication requirements describe what is required to communicate between the distributed components of the simulation and SAFIRE. Four, the testing required to ensure that SAFIRE still operates as expected after modifications and the addition of the RTI modules.

The following is a point form list of the requirements. Each of the areas is then expanded in their own section.

User Interface Requirements:

- Provide similar functionality to the current radar program.

P516249.PDF [Page: 13 of 101]

- May be disabled to keep D-SAFIRE running in real time.
- Perform any start-up and initialization procedures.

SAFIRE Modification Requirements:

- Modify SAFIRE to permit interaction with the RTI Communication Modules.
- Requires receiving a list of targets with position, velocity, acceleration and electronic counter measures status.
- The target information needed must be supplied through the RunTime Infrastructure (RTI) Communication Modules.
- The resulting radar status report must be published on the RTI.
- Execution of the radar simulation program should be controlled by the Radar Operator in the ACD.

RTI communication:

- This will provide interaction between the Radar Sensor (SAFIRE) and the RTI.
- The RTI will communicate with all other federates.
- Operational radar parameters (i.e. scenario) will be received from the RTI by the RTI communication modules communicated to the Radar Sensor
- The.simulation results will be gathered from the Radar Sensor and sent to other federate via the RTI.

Testing

- Facilitate testing in the absence of the ACD.
- Provide federated object interactions that would be maintained by other portions of the distributed simulation and communicate these through the Run-Time Infrastructure.
- Display all the federate objects and interactions.
- Provide a user interface for the testing.

1.2.1 User Interface Requirements

The user interface should provide similar functionality as compared to the current SAFIRE User Interface. While the program is part of a distributed simulation some of the input functions on the User Interface will have to be disabled. The input functions that are disabled will come from parts of the distributed simulation.

It is required that D-SAFIRE run in realtime to keep up with the realtime simulation in the ACD. To maximize the amount of time spent processing the radar returns it may be necessary to disable the GUI entirely. However, the GUI will be required for testing and debugging.

When SAFIRE runs as a stand alone application the User Interface must be fully functional. It will be necessary for the User Interface to know if it is running as a stand alone application or as part of the distributed simulation.

P516249.PDF [Page: 14 of 101]

At start up the user interface must detect if there is a distributed simulation in communication with the program. If there is a distributed simulation the user interface must join the simulation and disable the necessary inputs from the local interface. If there isn't a simulation the user interface must start up and execute as a stand alone application.

In an ideal situation SAFIRE and D-SAFIRE would be the same program. Having SAFIRE and D-SAFIRE the same program would depend on getting SAFIRE to run at least as fast as real time without sacrificing fidelity. If the fidelity of SAFIRE has to be compromised for the real time requirement then D-SAFIRE will be a separate program.

1.2.2 SAFIRE Modification Requirements

SAFIRE will receive the current target processing list from the ACD Simulation Environment through the RTI and the local RTI communication modules. This information should consist of the position, speed and acceleration of each simulated target relative to the Ownship. The target Aircraft data should also include the ECM settings and Electromagnetic (EM) characteristics (ie. Radar Cross Section and Swerling type). The radar model should also receive information regarding the radar mode and weapon configuration of the Ownship.

Given the current airspace state the radar sensor determines what hit information is available and returns the hit information to the Ownship for display. D-SAFIRE is an instance of a radar sensor which uses a radar model to realistically determine what targets (initially a maximum of 2) are detected from the current Ownship. The radar sensor will generate hit information consisting of the line of sight (LOS) range, LOS range rate, amplitude and type of each target detected by the radar model; as well as the current position of the antenna. This hit information is returned by the radar sensor to the Ownship. The internals of the Aircraft object are responsible for delivering the hit and radar information to the display.

Modifications to the current SAFIRE program will allow the local RTI communication modules to input and extract data from it. The latest target information and radar parameters will be recieved by the RTI modules and input into the program. The RTI communication modules will also be able to get the latest hit and radar information from SAFIRE in order to send the information to other parts of the simulation.

The simulation will be running in real time. Thus, D-SAFIRE must also run in real time. Modifications to the existing SAFIRE code were completed in order to have it run in real time or faster than real time, depending on the number of targets processed. The clock in the machine running D-SAFIRE will allow the radar sensor to remain synchronized with the real world time. There is no need to have each clock in the simulation set to the exact time. Each clock should advance at the same rate and this clock advancement will keep the simulation coordinated. There could be some problems with communication latency between machines. This will have to be

P516249.PDF [Page: 15 of 101]

investigated. The RTI does provide mechanisms for the controlled advancement of time. These mechanisms are not applicable to real time simulations and will not be needed.

The stability of the existing SAFIRE program over long periods of time is not known since SAFIRE was intended for short simulations and has never been tested over long periods of simulated time. After a period of time SAFIRE may need to be restarted to ensure that the software is running within known functional specifications. This operation must occur without operator intervention.

SAFIRE will only be required to operate in one mode (MPPRF-RWS) with a maximum of only two targets. This requirement is likely to be expanded in the future.

1.2.3 RTI Communication Requirements

The RTI communication modules are responsible for all communication between the Radar Sensor and the rest of the distributed simulation. The Radar Sensor will use the RTI communication modules to tell the RTI what information it is interested in receiving and what it can produce. The RTI will tell the Radar Sensor if any federate is interested in receiving output from the Radar Sensor. The information generated by the Radar Sensor will be sent to the RTI through the RTI communication modules. The RTI is responsible for delivering the information to all interested parties.

The RTI communication modules consist of an RTI Ambassador and a Federate Ambassador. Each federate has one RTI Ambassador and one Federate Ambassador. The federates use their RTI Ambassador to communicate with the RTI. The Federate Ambassador of a federate is used by the RTI to communicate with the federate. The RTI Ambassador is part of the RTI and can be used by instantiating an RTI Ambassador and using it to communicate with the distributed components. The Federate Ambassador is part of the federate. The Federate Ambassador has been defined by the RTI but it must be implemented as part of the federate.

The HLA RTI implementation to be used is the HLA RTI-NG 1.3 Version 3.1. (Where NG = Next Generation.) The HLA RTI 1.3 is currently proceeding through an IEEE standard approval process. The NG implementation represents a "from scratch" implementation of the HLA RTI 1.3 specification.

1.2.4 Testing Requirements

Testing is required to ensure that the D-SAFIRE program operates within the specification laid out in this document and that it communicates with the ACD in order to meet project objectives.

Test stubs will simulate the response (both initiating and receiving) of everything outside the radar sensor and allow D-SAFIRE to be tested without being connected to the ACD. The testing

P516249.PDF [Page: 16 of 101]

software will provide all information to the radar model which would normally be maintained by other parts of the distributed simulation. Communication between the radar sensor and the Test stubs will be through the RunTime Infrastructure (RTI). Thus, the testing stubs will require RTI communication modules.

A User Interface for testing is required to display all of the object interactions communicated across the RTI. This will capture all the communication between D-SAFIRE and the Test stub. The interactions will be displayed on the screen in a human readable form. The option of saving all the displayed interaction information to a file should also be implemented.

1.3 Conceptual Models

These models are included to illustrate graphically how the parts of the system are related. From these models more detailed models can be developed to demonstrate how the various parts will interact to accomplish specific goals or generate the expected behaviour of the system.

1.3.1 D-SAFIRE Model

This model (**Figure 1**)shows the separation of the components that will make up D-SAFIRE. The Graphical User Interface (GUI) and the Scientific Simulation Software are components of the exiting SAFIRE program. The RTI Communication component that will be needed is responsible for all communication with the other distributed components of the simulation. The RTI Communication component will have to interface with the GUI and SSS to insert parameters generated by the distributed components.

1.3.2 Aircraft Model

The conceptual model of the an Aircraft (see **Figure 2**) is intended to illustrate the relationship between an aircraft and the various properties and objects that make up an aircraft. The model extends only to the properties and attributes that are of concern in this project. The cardinality of each relationship is indicated on the figure.

An aircraft has four properties that are of interest to the radar model. The physical state of the aircraft: position, attitude, velocity and acceleration. The electromagnetic characteristics of the aircraft are used to determine the radar cross section and scintillation characteristics. The electronic counter measure configuration of the aircraft is used indicate the on/off status of any jamming of other radar sensors. D-SAFIRE will define and maintain the specific jamming technique being implemented for each aircraft. The Weapons Configuration is used for selection of certain operations of the local radar and display.

The aircraft also contains a radar sensor and a display. The display is used to take the information generated by the radar sensor and make the information human readable. The radar sensor

examines the current environment and returns information about any visible targets. The radar sensor delivers the hit information to the aircraft which in turn sends the information the display.

The scope of Call Up #3 of Contract W77140-0036 is only concerned with the radar sensor and the interactions between the radar sensor and the aircraft. Any aspects of the Display are outside the scope of this call up and in fact the entire contract.

1.3.3 RTI Communication Model

The conceptual model for the communication between the RTI and the Radar Sensor (or D-SAFIRE), the CF-18 ACD and the target simulator is illustrated in **Figure 3**. All communication between the components of the simulation must be passed through the RTI. The RTI communicates with each of the federates through their Federate Ambassadors and the federates communicate to the RTI through their RTI Ambassador. This communication includes all interactions and object updates the federates want to know about. All information available must be defined in the Federate Execution Details (.FED) file.

Each Federate indicates its readiness to receive or send information through the RTI. The RTI will deliver the information to all interested federates through their Federate Ambassador. When no federates are interested in a piece of information the federate that produces the information should refrain from sending that information. This will reduce the amount of wasted bandwidth on communication lines.

The Federate Ambassador is used by the RTI to deliver any information to the federate. The Federate Ambassador then delivers the information to the object in the federate. A Federate Ambassador is not allowed to "talk" to an RTI Ambassador. A concurrent access error will result if a Federate Ambassador tries to use an RTI Ambassador.

The exact structure of federates other than the D-SAFIRE federate may not be as illustrated in Figure 3. The scope of the this document is only the D-SAFIRE federation.

P516249.PDF [Page: 18 of 101]

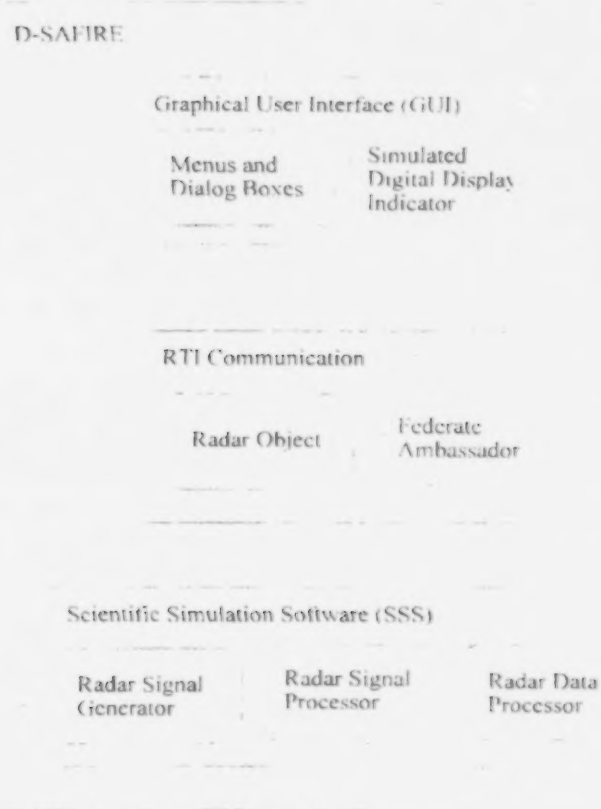

Figure 1 Model of D-SAFIRE components

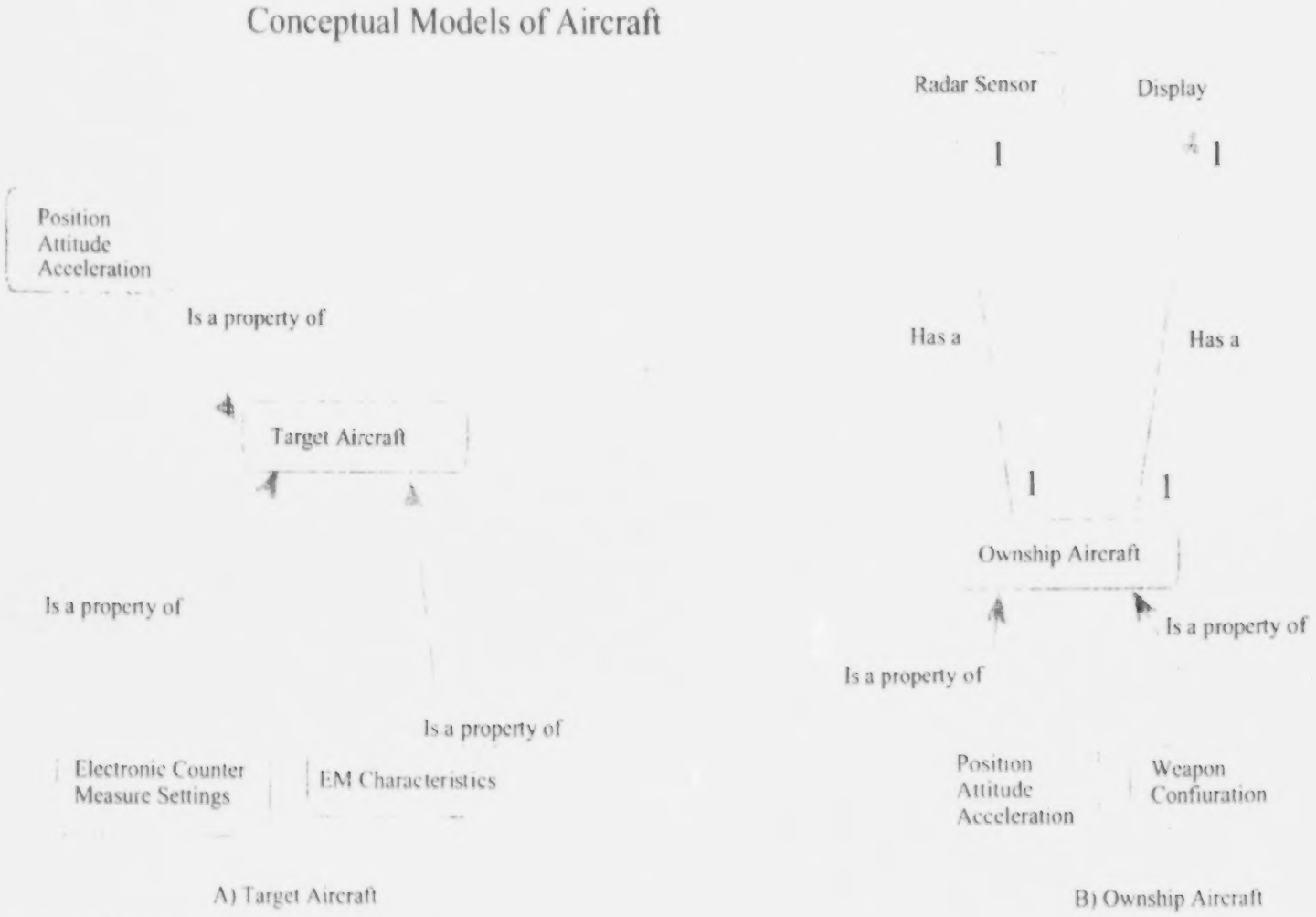

Figure 2 Conceptual Models of an Aircraft

Conceptual Model of RTI communication

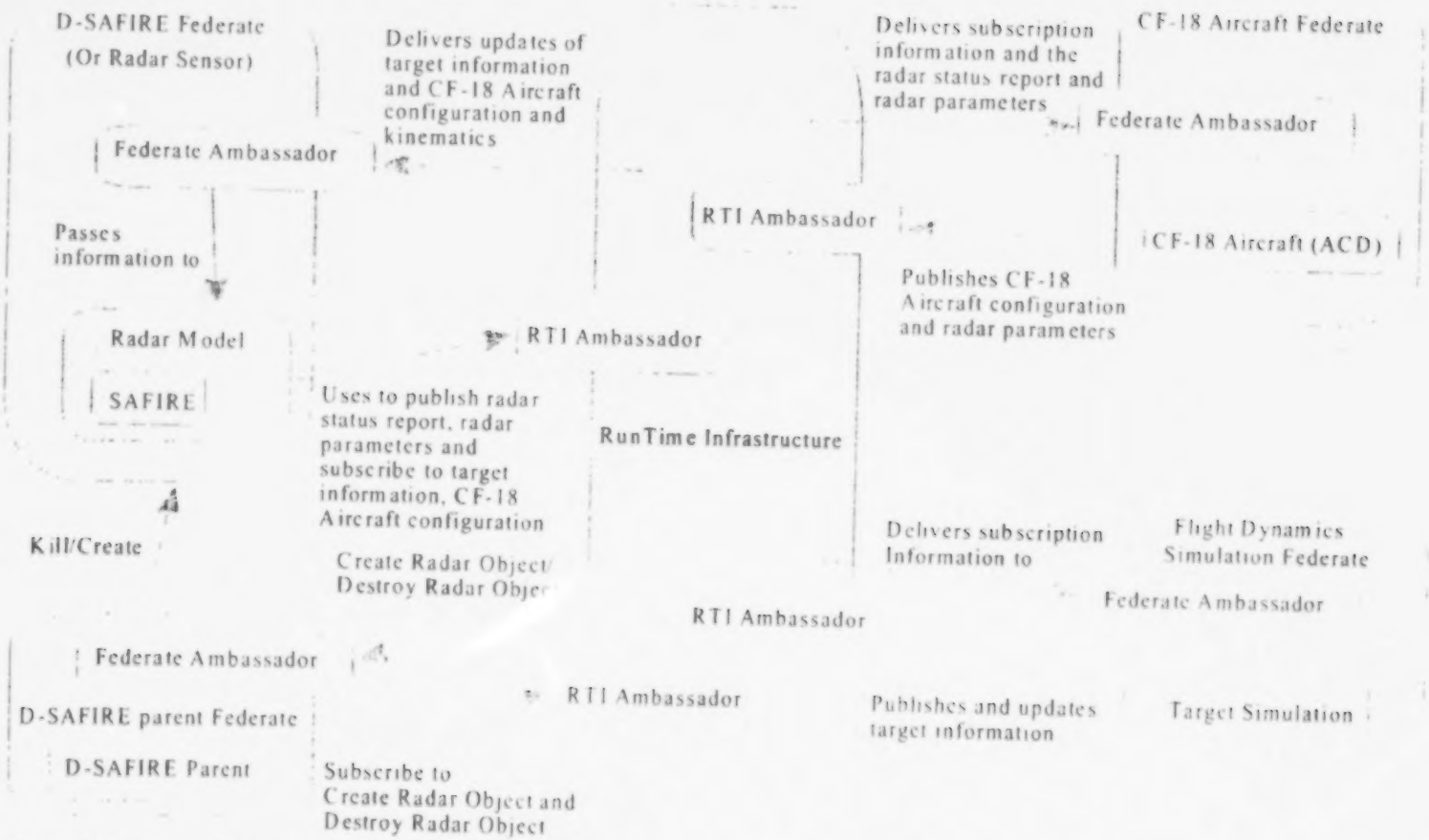

Figure 3 Conceptual Model of RTI Communication

P516245 PDF [Page: 21 of 101]

1.4 Operational Requirements

In this section the functions that the software needs to perform are examined and defined. As part of determining operational requirements a set of actors and use cases in which the actors play a role are defined. Each use case identifies a set of activities required to complete a process.

1.4.1 Actors

D-SAFIRE	This is a radar system. It receives information about all aircraft and generates a report which lists the target visible to the radar.
Run-Time Infrastructure	This is the interface between the Radar Sensor and all other components of the simulation.
All Interested Federates	This is all components interested in some piece of information
D-SAFIRE Radar Model	This is the part of D-SAFIRE that is responsible for analysing the current airspace state and producing a list of hits (or targets detected by the radar sensor).
D-SAFIRE Parent	A federate that has the power to create and destroy the D-SAFIRE federate.
Ownship	The aircraft that owns the radar sensor. This is the CF-18 Aircraft. (ACD)
Flight Dynamics Simulator	This generates the simulation environment. It produces a set of targets with a position, speed, acceleration, ECM setting, and EM configuration.
Radar Simulation Operator	A human responsible for start up operation of the radar component.

1.4.2 Use Cases

Use Case: Update Target Processing List
Actors: Flight Dynamics Simulator, RunTime Infrastructure (RTI), D-SAFIRE, D-SAFIRE Radar Model
Description:

1. The D-SAFIRE subscribes to updates of all target aircraft and Ownship in the simulation. (Update Target Processing List interaction class)
2. The Flight Dynamics Simulator submits updated target processing list to the RTI.
3. The RTI delivers the updated airspace state to D-SAFIRE through the Federate Ambassador

P516249.PDF [Page: 22 of 101]

4. D-SAFIRE takes the updated airspace state and uses it to generate the next set of hits as seen by the current platform.
   a. Perform a required transformation required to make the data readable by the D-SAFIRE Radar Model
   b. Insert the new data into the D-SAFIRE Radar Model at a suitable point.

The D-SAFIRE will continue to generate target information and send radar status reports based on the latest airspace state until an updated airspace state is received.

Use Case: Radar Status Report
Actors: RunTime Infrastructure (RTI), D-SAFIRE, D-SAFIRE Radar Model, All other interested federates
Description
1. D-SAFIRE completes a simulation of the scenario and produces a current hit list.
   a. The D-SAFIRE Radar Model completes a simulation of the current scenario and generates a new hit list.
   b. If there is a new hit list then D-SAFIRE generates a new radar status report.
2. D-SAFIRE publishes the new radar status report on the RTI, through its RTI Ambassador
3. The RTI delivers the updated radar status report to all interested parties.

Use Case: Start-Up
Actors: RunTime Infrastructure (RTI), D-SAFIRE, Radar Operator, D-SAFIRE Radar Model, D-SAFIRE Parent
Description
1. The radar operator starts the RTI exec
2. The radar operator starts the D-SAFIRE Parent federate
3. The D-SAFIRE Parent joins the federation and waits to recieve a signal to Create Radar Object.
4. When the Create Radar Object signal arrives the D-SAFIRE Parent starts D-SAFIRE
5. D-SAFIRE associates itself with the calling Ownship and subscribes to the Radar Parameters of the Ownship, and Target Processing List.
6. When the initial Target Processing List, Ownship attributes and radar parameters are received D-SAFIRE translates the scenario into a form readable by the D-SAFIRE Radar Model.
7. D-SAFIRE waits for the Radar Continue Request.
8. When the Radar Continue Request arrives D-SAFIRE starts the D-SAFIRE Radar Model with the current Target Processing List, Ownship properties and radar parameters.

Use Case: Radar Parameter Change
Actors: RunTime Infrastructure (RTI), D-SAFIRE, D-SAFIRE Radar Model, Ownship
Description

PS16249.PDF [Page: 23 of 101]

1. Ownship sends a parameter change to the D-SAFIRE through the RTI.
2. The RTI delivers a Radar Parameter Change to D-SAFIRE (via its Federate Ambassador)
3. D-SAFIRE transforms the Radar Parameter Change into something readable by the D-SAFIRE Radar Model.
4. D-SAFIRE inserts the parameter change into the radar simulator at a suitable point of operation if the parameter is within an acceptable range.
5. D-SAFIRE sends a confirmation of parameter change back to the CF-18 ACD through the RTI Ambassador.

Use Case: Radar Pause Request

Actors: RunTime Infrastructure (RTI), D-SAFIRE, D-SAFIRE Radar Model

Description

1. The RTI delivers a Radar Pause Request to the Federate Ambassador of the D-SAFIRE.
2. D-SAFIRE halts the radar simulator while maintaining the current state of the radar simulation.

Use Case: Destroy Radar Object

Actors: RunTime Infrastructure, D-SAFIRE

1. The RTI delivers a Destroy Radar Object to the Federate Ambassador of D-SAFIRE Parent.
2. D-SAFIRE Parent kills D-SAFIRE.

P516249.PDF [Page: 24 of 101]

2. Software Design

2.1 Flow of Execution

This section is intended to deal with how the new software modules should fit into the flow of execution. There are three options in this case. One, we could execute the RTI communication software after the Graphical User Interface (GUI) but before the Scientific Simulation Software (SSS). Two, we could execute the RTI communication modules after the SSS but before the GUI. Finally, we could have a combination of both. These are all illustrated in **Figure 4**

The advantage of executing the RTI modules after the GUI and before the SSS (**Figure 4B**) is that we can put the most current data into the SSS before it executes. In this case the data used for the radar status report might be a little stale, since the GUI executes before the control returns to the RTI modules. Once control returns to the RTI modules, the data from the SSS then be published. This delay shouldn't be too much of a factor since most of the execution time is in the SSS.

Having the RTI software execute after the SSS (**Figure 4C**) would allow the most up to date data to be available for publication. In this case, the latest target list and radar parameters would be inserted into the SSS. Control would return to the GUI before the SSS. The SSS it could use the new data once control returned from the GUI. This might mean that the target and parameter information would be stale before it is processed

The third alternative would be to execute the RTI modules both before and after executing the SSS (**Figure 4D**). In this case when the RTI module is executed before the SSS it would be responsible for inserting the latest target data and parameters into the SSS. If there is no new information then the RTI modules could wait for new target data before letting the SSS run or let the SSS run with the old data. When the RTI module is executed after the SSS it would publish the latest radar status report and the parameters. This method of execution involves some additional complication on the part of the calling RTI module knowing what should be done when it is called. This can be done by implementing a flag that the RTI software would set when finished the current task.

With the GUI in the control path it will be necessary to disable some of the GUI's functionality. This will prevent modification of simulation parameters through the user interface. This can be done through the use of flags and compiler directives to skip certain sections of code.

In order to keep the performance of D-SAFIRE as close to realtime as possible the GUI should be eliminated from the execution loop (**Figure 4E**). In this case control cycles between the RTI modules and the SSS. Each module will have the most recent information. However, it is necessary to have the GUI functioning for debugging and testing purposes. As a result there will be two versions of D-SAFIRE, one with a functioning GUI and a control flow as illustrated in **Figure 4D**, another with the GUI removed and a control flow as illustrated in **Figure 4E**.

P516249.PDF [Page: 25 of 101]

Figure 4 Control flow Diagram

P516249.PDF [Page: 26 of 101]

An Algorithm for **Figure 4D**

```
Bool: Do_Receive = TRUE // initial value

Radar_Object::Run()
{
        if (Do_Receive)
        {
        /* Check for new data and insert it into the SSS */
                :
        /* done receive stuff */
                Do_Recieve = FALSE

        }
        else
        {
        /* get the latest radar status and sent it */
                :
        /* done sending */
                Do_Recieve = TRUE
        }
}
```

The following are modifications to the existing SAFIRE code that would support the control flow illustrated in **Figure 4D and E**. The modifications are highlighted in ***bold italics*** print.

Modifications to SAFIRE
File: safire.c

```
/*
 * main for application safire
 */
int
main(
   int argc,
   char **argv
)
{
                :

 /* if full distributed mode then jump into the simulation and do not use the
    GUI */
#if defined(DSAFREM)
   printf("\n\n");
   printf(" ********************************************************\n");
   printf(" *  Distributed SAFIRE Version 2.6    May 1999  *\n");
   printf(" ********************************************************\n\n");
   safire_.prog_start=TRUE;
   safire_.replay=0;

   safire_init_();
```

P516249.PDF [Page: 27 of 101]


```
   rti_init();
   sim_cont_.SIM.LOCAL = 0;
   while(1)
   {
    rti_run();
    sss_control_();
    rti_run();
   }
#else

// used compiler directive to include RTI and C++ components only for the
// distributed simulation
#if defined(DSAFIRE)     // distributed simulation
   printf("\n\n    Distributed SAFIRE\n");
   sim_cont_.SIM.LOCAL = 0;
   rti_init();
   screen_start((Widget)NULL,(XtPointer)NULL,(XtPointer)NULL);
#else
          printf("  Range     Velocity   Type    Azimuth    Elevation\n");

   sim_cont_.SIM.LOCAL = 1; // local simulation only
#endif

#endif   /* end if defined(DSAFREM) */
   return 0;
}
```

File: safire_stubs.c

```
Function. screen_simulate()
{
 long i,j=0;

 /*if(time_data_.radar_run_time >= time_data_.radar_stop_time)*/
 if(time_data_.radar_run_time >=
   seg_.OWNSEG[sim_cont_ SIM.NUM_OF_OWN_SEG-1].END_TIME)
 {
  if(Batch_mode)
   exit(0);

  if(!safire_.replay)
  {
   running=FALSE;
   printf("Scenario has finished. Press RESET to reset the scenario.\n\n");
   return;
  }
 }
```

```
 /* Process one phase */
 if(safire_.replay)
 {
  /* read data for phase from sim_disp.dat here ? */
  get_replay_data();
 }
 else
 {
#if defined(DSAFIRE)
        Radar_Object.run()
        sss_control_();
        Radar_Object.run();
#else
        sss_control_();
#endif
 }

 /* Following line works for SAFIRE (fast version). Timing Bug ? (3/10/95) */
 printf(" \r");
                :
}

void
screen_reset(
   Widget widget,
   XtPointer clientData,
   XtPointer callData
)
{
   /*** DTB_USER_CODE_START vvv Add C variables and code below vvv ***/
   /*** DTB_USER_CODE_END   ^^^ Add C variables and code above ^^^ ***/

   /*** DTB_USER_CODE_START vvv Add C code below vvv ***/
  /* do not compile for the debug/testing version fo D-SAFIRE */
#if !defined(DSAFGUI)
   if(running) running=FALSE;
   reset_flag=TRUE;
   printf("\n\nScreen reset\n");
   erase=1; /* erase targets */

   display_background(pix_1);
   screen_initialise();

   show_clutter_and_jammer_and_antel_status();
   clear_twsplot();
#endif
   /*** DTB_USER_CODE_END   ^^^ Add C code above ^^^ ***/
}
```

Radar Object Code
File: Radar_Object.cpp

```
bool Do_Receive;
bool Do_Send;

/* the method called from the GUI to handle the RTI communications */
void Radar_Object::Run()
{
        /* receive new target data and radar parameters */
        if (Do_Receive == TRUE)
        {
                :
                Do_Recieve = FALSE;

        }
        /* pulish the leatest radar returns and parameters used */
        else
        {
                :
                Do_Receive = TRUE;

        } /* end if */
} /* end run */
```

2.2 Radar Create Request and Radar Destroy Request

One of the requirements is to be able to terminate the execution of the D-SAFIRE program and restart D-SAFIRE. The D-SAFIRE program will take care of stopping it's own exeuction when it receives a Radar Destroy Request. Since the D-SAFIRE program is executed in a separate shell as a child process the parent process will ensure that the shell process terminates as well. The parent process will also join the distributed simulation. As part of the simulation the parent can listen for requests to destroy and create the D-SAFIRE process. The parent program will also listen for the pause, continue and radar parameters. This will allow the parent to restart D-SAFIRE in it's previous state.

The stability of the existing SAFIRE program over long periods of time has never been tested and cannot be guaranteed. A restart, either by a request from a component of the distributed simulation, or after a period of time can be implemented in the parent. The restart after a maximum operating time can be implemented as a timer. When the timer goes off the parent will automatically kill the D-SAFIRE process, restart D-SAFIRE. The parent will then send it the previous parameters and either a pause or continue. This will allow D-SAFIRE to restart and resume from the previous state. When a destroy request is received the parent will kill the D-SAFIRE process. The parent will wait for a create request before the D-SAFIRE process will be created.

The state of the ownship aircraft and targets will not be saved on a destroy request. The state can be recovered from the simulation environment on the next update, which should occur at a rate of

P516249.PDF [Page: 30 of 101]

30Hz. The create and destroy requests may be used to associate the radar with another aircraft. In this case it would not be beneficial to save the state since the perspective will have changed. The updates should come at a high enough rate that D-SAFIRE could reestablish the current state in a much shorter time than it takes to destroy and create the new D-SAFIRE process. If it is necessary to save certain values then D-SAFIRE could be made to save current state before exiting.

In order to keep D-SAFIRE running in real time, the majority of the processor time should be devoted to executing D-SAFIRE, the parent process should sleep during most of the inactive periods. The parent will have to periodically check for incoming interactions from the RTI and the timer.

2.2.1 D-SAFIRE Parent

The main() function of the D-SAFIRE parent starts everything. This function will create a DSAFIRE_Parent object. The DSAFIRE_Parent will first set up RTI communications for itself and then wait for an interaction to request the creation of D-SAFIRE. If the D-SAFIRE parent is unable to find an RTIexec then the D-SAFIRE parent should exit.

The D-SAFIRE Parent process will have to monitor the RTI for Radar Create Request, Radar Destroy Request and Terminate Simulation interactions as well as the duration of the current execution. In order to keep execution stable, if D-SAFIRE has been running for the maximum allowable time; this process will kill and restart a D-SAFIRE process. The D-SAFIRE Parent process will send a signal to indicate to D-SAFIRE that a kill signal is coming. This will give D-SAFIRE time to save any necessary information.

Restarting D-SAFRIE will require that the Parent maintain some information on the state of D-SAFIRE. The ACD will not know to send interactions in order to have D-SAFIRE start in the correct state. This information includes the paused/running state as well as the radar parameters being used. This information will be sent by the parent when it restarts D-SAFIRE. This information will also be maintained after a destroy interaction. This information will be sent when a create interaction is sent. This should result in a faster restart of D-SAFIRE.

The Terminate Simulation interaction is used to tell the D-SAFIRE Parent that the simulation is finished. The D-SAFIRE Parent will then kill the D-SAFIRE process and exit itself.

Definition of the main function for the D-SAFIRE Parent:

```
void main(int argc, char **argv)
{
        DSAFIRE_Parent  D_SAF_Parent;

        D_SAF_Parent.init();

        D_SAF_Parent.run();
}
```

P516249.PDF [Page: 31 of 101]

The definition of the D_SAFIRE_Parent class can be seen in Appendix A.

The instructions for making the D-SAFIRE parent program (psafire & psafired) have been included in the makefile for the SAFIRE project. The difference between psafire and psafired is the version of D-SAFIRE that is started when a creat radar object interaction is received. The program psafire starts the version of D-SAFIRE that has the GUI removed (dsafire). The program psafired starts the version of D-SAFIRE with the GUI (dsafired).

The D-SAFIRE parent starts D-SAFIRE by executing a shell script in a new process. The use of a shell script allows the output of D-SAFIRE to be redirected to a file without having to modify and recompile the D-SAFIRE Parent program. Since the output of both the D-SAFIRE parent program and D-SAFIRE will be on the same console it is likely that the user will want to redirect the output of D-SAFIRE.

2.2.2 D-SAFIRE Parent's Federate Ambassador

The methods of the Federate Ambassador are called by the RTI to deliver the Terminate_Simulation, Radar_Create_Request and Radar_Destroy_Request interactions. The methods are defined by the RTI. Not all methods need to be implemented. Here are the functions that will need to be supported.

D-SAFIRE Parent Federate Ambassador supported methods

```
/* This call back informs the federate of an interaction in the federation that is relevant to the federate's current
subscription interests */
void Parent_Federate_Ambassador::receiveInteraction (
      RTI::InteractionClassHandle theInteraction,
      const RTI::ParameterHandleValuePairSet& theParameters,
      const RTI::FedTime& theTime,
      const char  *theTag,
      RTI::EventRetractionHandle theHandle)
throw (
  RTI::InteractionClassNotKnown,
  RTI::InteractionParameterNotKnown,
  RTI::InvalidFederationTime,
  RTI::FederateInternalError)

/* This call back informs the federate of an interaction in the federation that is relevant to the federate's current
subscription interests */
void Parent_Federate_Ambassador::receiveInteraction (
      RTI::InteractionClassHandle  theInteraction,
      const RTI::ParameterHandleValuePairSet& theParameters,
      const char* theTag)
throw (
  RTI::InteractionClassNotKnown,
  RTI::InteractionParameterNotKnown,
  RTI::FederateInternalError)
```

P516249.PDF [Page: 32 of 101]


```
/* this advises the D-SAFIRE parent of active subscribers for an interaction class */
void turnInteractionsOn (
  RTI::InteractionClassHandle theHandle) // supplied C1
throw (
  RTI::InteractionClassNotPublished,
  RTI::FederateInternalError);

/* this advises the D-SAFIRE parent of the absence active subscribers for an interaction class */
void turnInteractionsOff (
  RTI::InteractionClassHandle theHandle) // supplied C1
throw (
  RTI::InteractionClassNotPublished,
  RTI::FederateInternalError);

/* this method tells the federate ambassador where to deliver objects and interactions */
void Parent_Federate_Ambassador::Set_Federate(SAFIRE_Parent *A_Federate)
```

2.3 User Interface

As was discussed in section 1.2.1 the user interface for D-SAFIRE may be eliminated in order to keep the program running in realtime but will have to remain for debugging and testing. When the GUI is being used it will be unchanged for the most part. The look and feel of the User Interface will be the same. Some of the input that would normally come from the User Interface will come via the RTI. The changes will involve disabling some of the inputs from the User Interface while functioning as part of the distributed simulation.

It may be suitable to use a boolean variable to indicate which mode SAFIRE will operate in (distributed or local). Each modification will be encapsulated by a test of the condition of this variable. Alternatively if modifications are required to create a faster than real time implementation of SAFIRE for D-SAFIRE then it would be it's own entity and not will be run as a local simulation

Removing the graphical user interface part of D-SAFIRE is implemented through compiler directives. This will direct the compiler to compile specific sections while skipping other sections. These directives will be defined in the makefile for the version with and without the GUI.

The exact changes required will be examined in the following sections. As each specific interaction is analysed the necessary changes will be defined.

2.3.1 Disabling Local Control

As part of a distributed simulation we need to disable local changes to the parameters. Any requests to change the parameters will be communicated through the distributed simulation. The capability of disabling local control can be implemented through a boolean variable (Local).Local will be set to false to indicate that the program is part of a distributed simulation.

P516249.PDF [Page: 33 of 101]

This variable is added to the sim_cont_.SIM structure and the SIM common block. The sim_cont_.SIM structure is defined in the F_sim.h file and the SIM common block is defined in the file sim.inc.

The implementation details of disabling local control of specific functions will be discussed later in this document in sections dealing those specific functions.

Boolean: Local
If able to establish distributed simulation
    Local = False
else Local = true

Modification to SAFIRE

File: safire.c

```
/*
 * main for application safire
 */
int
main(
   int argc,
   char **argv
)
{
                    :
#if defined(DSAFIRE)     // distributed simulation
   printf("\n\n   Distributed SAFIRE\n");
   sim_cont_.SIM.LOCAL = 0;
   rti_init();
   screen_start((Widget)NULL,(XtPointer)NULL,(XtPointer)NULL);
#else
        printf("  Range    Velocity   Type   Azimuth   Elevation\n");

   sim_cont_.SIM.LOCAL = 1; // local simulation only
#endif

                    :

}
```

File: sim.inc

```
* FILE NAME   SIM.INC
*
* CONFIGURATION IDENTIFIER:  SSS-U-SIM.INC
*
* REVISION HISTORY:
*
*  DATE     AUTHOR        COMMENTS
*  ----     ------        --------
*  04/01/94  A. WALKER     BASELINE 1 COMPLETED
*  15/11/95  G.W  Geling   BASELINE 2a COMPLETED
*  08/01/96  G.P. Currie   Added NEW_PHASE to SIM structure
```

P516249.PDF [Page: 34 of 101]


```
*  14/08/96  G.P. Currie    Used alternate form of PARAMETER statement so that
*                           common structures can be used with C.
*
C This file contains variables related to the general control of
C the simulator.
C This file should always be included before all the others
C because it contains parameters used by other include files.
C-----------------------------------------------------------------------


C GPC
      INTEGER MAX_NUMTARG, MAX_NUMSEG
      PARAMETER  (MAX_NUMTARG=10, MAX_NUMSEG=10)

      STRUCTURE /SIMULATION_CONTROL/

        LOGICAL   CLUTTER_ON
        LOGICAL   VFO_ON
        LOGICAL   CDE_ON
        LOGICAL   GUARD_CHAN_IND
        LOGICAL   BUFFER_IS_FULL
        LOGICAL   NEW_MODE_IND
        LOGICAL   FIRST_TIME
        INTEGER   NUMTARG
        INTEGER   NUMJAM
        INTEGER   NUM_OF_TARG_SEG( MAX_NUMTARG )
        INTEGER   NUM_OF_OWN_SEG
C The phase number we will use in NOISE and in RECEIVER_BLANKING.
        INTEGER   PHASE_NO
        LOGICAL   NEW_PHASE
C Time duration of previous phase
        REAL      PREV_PHASE_TIME
        REAL*8    TIME
        LOGICAL   BORESIGHT_ON
        LOGICAL   NEW_PRF_IND
        LOGICAL LOCAL

      END STRUCTURE

      RECORD /SIMULATION_CONTROL/SIM

      COMMON /SIM_CONT/SIM
```

File: F_sim.h

```
/*****************************************************************************
```

Code date: 25 Jan 1996

```
Version:    1.0

Edit History:

Date            Author          Comments
----            ------          --------
25 Jan 1996     G.P. Currie     Initial creation

------------------------------ USAGE ----------------------------------

#include "F_sim.h"

---------------------------- DESCRIPTION --------------------------------

Declaration of external FORTRAN common 'sim.inc' data containing
general simulation variables for transfer between C and FORTRAN.

*****************************************************************************/

extern set_new_mode_ind_();

#define MAX_NUMTARG 10
#define MAX_NUMSEG  10

struct simulation_control
{
 int  CLUTTER_ON;
 int  VFO_ON;
 int  CDE_ON;
 int  GUARD_CHAN_IND;
 int  BUFFER_IS_FULL;
 int  NEW_MODE_IND;
 int  FIRST_TIME,
 int  NUMTARG,
 int  NUMJAM;
 int  NUM_OF_TARG_SEG[MAX_NUMTARG],
 int  NUM_OF_OWN_SEG;
 int  PHASE_NO;              /* The phase number we will use in NOISE
                                and in RECEIVER_BLANKING. */
 int  NEW_PHASE;
 float PREV_PHASE_TIME;  /* Time duration of previous phase */
 double TIME,
 int  BORESIGHT_ON;
 int  NEW_PRF_IND;
 int  LOCAL;
};
```

P516249.PDF [Page: 36 of 101]


```
extern struct SIM_CONT_type
{
 struct simulation_control SIM;
};

extern struct SIM_CONT_type sim_cont_;
```

2.4 Radar Object Design

This object will control the interface between the RTI and the SAFIRE program. The radar object is responsible for receiving any parameter changes to SAFIRE, any updates of target or ownship status, and having those changes input into the SSS. The Radar Object will also control publishing the status report and current parameters. In this section the Radar_Object class methods and attributes will be defined.

This section will analyse each of the RTI interactions that D-SAFIRE will have to handle. The analysis will be in the form of what is required of the Radar Object and what modifications to the existing SAFIRE code are necessary. The pseudo-code required for the Radar Object to fulfill the requirements of the interaction will be presented in short segments within the subsection dealing with that requirement. The final subsection, 2.4.7, will present a summary of the pseudo-code and method definitions for the Radar Object. This will give a coherent presentation of the Radar Object.

2.4.1 Update Target Processing List

This section describes how the Radar Object will get the latest information on all target aircraft into SAFIRE. The description includes code required in the Radar Object as well as modifications to the SAFIRE code.

The method for receiving interactions will test for the Update_Target_Processing_List interaction. A Radar_Object method is defined to specifically handle all arriving interactions. The method tests for the type of interaction, extracts the data and gives the data the object capable of handling the data.

```
/* the method called from the GUI to handle the RTI communications */
void Radar_Object::Run()
{
                    :
/* tick the RTI ambassador to deliver any interactions
   The first parameter is the minimum time for the RTI Ambassador to take
   The second  paramter is the maximum time for the RTI Ambassador
   During this time the RTI shoudl deliver any updates to the targets, the ownship and radar parameters */
        My_RTIAmb.tick(0.1,1.0);

                    :

        if (Do_Receive)
```

P516249.PDF [Page: 37 of 101]


```
        {
                /* char forl an update of the targets and ownship */
                If (TargetUpdateReceived == TRUE)
                {
                        /* Insert the updated targets into SAFIRE */
                        Target_List.update();
                }

                If (AircraftUpdateReceived == TRUE)
                {
                        /* insert the new ownship into SAFIRE */
                        Aircraft.update();
                }

                /* set the receive/send flags for sending next time */
                Do_Recieve = FALSE;

        } /* end if */
        else
        {
                :
} /* end run */

/* A method for processing this interaction will be called from the federate ambassador */
void Radar_Object::recieveInteraction(RTI::InteractionClassHandle theInteraction)
{
        /* check what kind of interaction was received */
                :
        else if (theInteraction == UpdateTargetProcessingListInteraction)
        {
                Get the parameter handle
                check for expected parameter handle
                extract parameter from the interaction
                set the target list to the new values
        }


                :
}
```

2.4.1.1 Class Descriptions

Classes and data structures were developed to handle the specific components of a target update. The content of a target update is defined in the D-SAFIRE Federation Object model (FOM) [1]. A Target_List class was defined to coordinate the reception of new target data and inserting the data into the SSS part of SAFIRE. The target data is received as an array of structures that hold the data for a single target. The Target_List maintains the target data as an array of Target_Status objects.

A Target_Status class was defined to handle the target data for a specific target. This keeps the

P516249.PDF [Page: 38 of 101]

details of the target data hidden from the specifics of the list implementation. The components of the Target_Status are defined in Table 1. Two of the components are compound data types (ecm_cnf and en_cnf). These components are defined as objects in the Target_Status class. The objects will handle the representation of these data types.

Table 1 Target Data

Type	Field		Description
integer	ID		Unique identifier for the target
boolean	visible		Indicates if the target's radar return is to be processed by SAFIRE
	ecm_cnf	jammer_on	Indicate if the target's jammer is on or off
	em_cnf	target_rho_magnitude	Radar cross sectional area
		target_rcs_model	Swerling model
float	range		Range from Ownship aircraft
float	radial velocity		Radial velocity with respect to Ownship aircraft
float	radial acceleration		Radial acceleration with respect to Ownship aircraft
float	az		Azimuth angle from Ownship
float	el		Elevation Angle from Ownship

The ecm_cnf and em_cnf components of the Target_Status are defined as a class in order to keep the representation consistent with the FOM. The ecm_cnf class maintains the jammer_on status. The em_cnf class contains the electromagnetic configuration of the target. The electromagnetic configuration consists of the radar cross sectional area and the model to be used in simulating a fluctuation of the radar cross sectional area.

Modification to SAFIRE code:
File: rsg.f
Subroutine PROFILE_GENERATOR(DEL_TIME)

```
C ************************************************************
C Check to see if this is a local or distributed simulation If it is distributed then the target and ownship
C updates will come from the distributed components. The updates to the target profile will have already been
C done. If it is local then use the profile generator.
C
   IF (SIM.LOCAL) THEN
C
```

PS16249.PDF [Page: 39 of 101]


```
C Do initialization, otherwise do normal procedure
C     IF (SIM.INITIALIZE) THEN
      IF (SIM.FIRST_TIME) THEN
                    :
C ***************************************************************************
C End of routine procedure
C ***************************************************************************
      END IF
    ENDIF

SUBROUTINE TRANSMIT_ANTENNA
                    :
C test for local or distributed simulation
C if this is a dsitributed simulation then we can use a simplified version
C of this subroutine
          IF (SIM.LOCAL) THEN
                    :

          ELSE

C distributed simulation which means we can do a simplified version
                    :

          END IF
```

2.4.2 Radar Parameters

In this section each of the radar parameters will be examined. The parameters we are concerned with are defined in the Federated Object Model. The examination of each parameters will consist of what SAFIRE variables need to be set and what modifications to SAFIRE behaviour are necessary and how the radar object will support receiving parameters via an RTI interaction and inserting the parameters in to SAFIRE.

One of the concerns in the design is how to disable inpút from the user interface of D-SAFIRE, when the GUI is enabled. This will not be a concern if the GUI is completely disabled as is proposed once the testing and debugging is complete. The Radar_Object must keep a local copy of the parameter values. The local copy of the parameters can then be compared to the parameters in SAFIRE. Any modifications to the parameters by the GUI component of SAFIRE can be undone and thus prevent any local changes to the radar parameters. Any additional modifications to SAFIRE for a specific parameter will be define in the subsection that deals with that parameter.

The radar parameters should be sent back to the ownship at a rate of 30Hz. The radar parameters interaction will be both sent and received. This will allow D-SAFIRE to confirm or deny radar parameter changes that come from the Ownship Aircraft.

```
/* the method called from the GUI to handle the RTI communications */
void Radar_Object.:Run()
```

PS16249 PDF [Page: 40 of 101]


```
{
        if (Do_Receive)
        {
                /* tick the RTI ambassador to deliver any interactions
                   The first parameter is the minimum time for the RTI Ambassador to take
                   The second  paramter is the maximum time for the RTI Ambassador
                During this time the RTI should deliver any updates to the targets, the ownship and radar
                parameters */
                 My_RTIAmb.tick(0.1,1.0);

                /* having the received radar parameters update the radar parameters is safire every time will
                   allow the received parameters to overwrite any input from the GUI */
                Received_Radar_Parameters.upddate();
                Do_Recieve = FALSE;

        }
        else
        {

                /* send the radar parameters */
                Send_Radar_Parameters();

                /* Radar Status report */
                Send_Radar_Status();

                Do_Recieve = TRUE;

        }
}

/* A method for processing this interaction will be called from the federate ambassador */
Radar_Object::recieveInteraction(RTI::InteractionClassHandle theInteraction)
{
                :
        if (theInteraction == Radar_Parameter_Interaction)
                Get the parameter handle
                check for expected parameter handle
                extract parameter from the interaction
                set the parameters to the new values
                :
}


/* this method will build the Radar_Parameter_Change interaction and send it */
void Radar_Object::Send_Radar_Parameters()
{
        /* build the Radar_Parameter_Set object */
        Radar_Parameters A_Set_Of_Radar_Parameters();
        A_Set_Of_Radar_Parameters.get_radar_parameters();
```

P516249.PDF [Page: 41 of 101]


```
        /* build the interaction class and parameter - value pair sets */
                ⋮

        /* send the interaction */
                ⋮
}
```

2.4.2.1 Class Descriptions

The Radar Parameter class is developed to handle the transfer of the radar parameters between the RTI and the SSS part of SAFIRE. The radar parameters will be both sent and received via the RTI. Any changes requested by the Air Crewstation Demostrator will be set to D-SAFIRE as an interaction. The requested change of parameters will be processed by the Radar_Parameters object. One of the processing steps will be to ensure that the requested change is within capability of D-SAFIRE before applying the change.

The Radar_Parameters object will contain all the parameters defined in the Federation Object Model (FOM). A list of parameters can be found in Table 2. Most of the parameters are simple data types, however, the acq_scale parameter consisted of two parameter so a Scale class was defined to maintain the acq_scale paramters and the Radar_Parameters object conatins a Scale object. The radar_mode_ind is defined as an enumerated type. The type is just an enumeration of the possible radar modes. The acquisition mode parameters (TAGC, acq_bump, acq_az, acq_el, acq_vel, acq_range, acq_scale) are beyond the scope of this design

Table 2 Radar Parameters

Type	Parameter	Description
enum	radar_mode_ind	Indicates the mode of radar
integer	radar_numbar	Number of scan bars
float	radar_az_center	Current azimuth centre of search pattern
float	radar_el_center	Current elevation centre of the search pattern
float	radar_del_az	Current half width of the search pattern
boolean	radar_acq_ind	Indicates radar sub mode acqusition
float	TAGC	Total automatic gain control
boolean	acq_bump	STT target re-selection indicator
float	acq_az	Acquisition centre azimuth search volume
float	acq_el	Acquisition centre elevation search volume
float	acq_vel	Acquisition centre velocity search volume

P516249.PDF [Page: 42 of 101]

float	acq_range		Acquisition venter range search volume
Scale	acq_scale	rdelta	Acquisition range extent parameter
		vdelta	Acquisition velocity extent parameter
float	radar_az		Current antenna azimuth position relative to ownship heading
float	radar_el		Current antenna elevation position relative to ownship heading

2.4.2.2 Radar_Parameter_Change.radar_mode_ind

Since the scope of this call-up is for SAFIRE to operate in MPRF-RWS and MPRF-TWS. We only need to design changes necessary to facilitate changes between these two modes. A preliminary design for supporting additional modes is presented in Appendix B.

Code for Radar_Parameters

```
include safirefc.inc, radar.inc, sim.inc

void Radar_Parameters::update_parameters()
{
                    :
/* check if the radar mode has changed */
        if (radar_mode_ind ≠ radar_.RADAR.MODE_IND)
        {
                If Radar_.RADAR.MODE_IND = RWS_HPRF_MODE
                        New_prf_ind = true
                        New_prf_mode = PRF_RWS
                else
                        New_mode(RWS)
        }
                    :
}

/* this method reads the parameters that the SSS has used to perform the current simulation and builds the
Radar_Parameters object */
void Radar_Parameter_Set::get_Radar_Parameters()
{
                    :
        radar_mode_ind = radar_.RADAR.MODE_IND
                    :
}
```

Modifications to SAFIRE

No modifications are anticipated to support a change in the radar_mode_ind.

P516249.PDF [Page: 43 of 101]

2.4.2.3 Radar_Parameter_Change.radar_numbar

Code for Radar_Parameter_Set

```
include safirefc.h, F_radar.h, F_sim.h

void Radar_Parameter_Set::update_parameters()
{
                    :
        if radar_numbar != radar_.RADAR.NUMBAR
        {
                // verify an appropriate number of scan bars
                switch(rdr_prm.radar_numbar)
                {
                case 1  case 2: case 4: case 6:;
                        antenna_data_.new_number_of_scanbars = radar_numbar
                        radar_.RADAR.NEW_BAR_NUM_IND = True
                        safire_.change_flag = true
        }               break;
                :
}

/* this method reads the parameters that the SSS has used to perform the current simulation and returns the
parameters as a Radar_Parameter_Set */
void Radar_Parameter_Set::get_Radar_Parameters()
{
                    :
        radar_numbar = radar_.RADAR.NUMBAR
                    :
}
```

Modifications to SAFIRE

No modifications to existing SAFIRE code are anticipated to support a change in radar_numbar.

2.4.2.4 Radar_Parameter_Change.radar_az_center

This parameter represents the azimuthal center of the radar scan range. The maximum scan width for SAFIRE is 70 degrees

Code for Radar_Parameter_Set

```
include safirefc.h, F_radar.h, F_sim.h


void Radar_Parameter_Set::update_parameters()
{
                    :
        if radar_az_center = Radar_ RADAR.AZ_CENTER
        {
                // verify that the value is between -70 and 70 degrees
                if ((rdr_prm.radar_del_az <= MAX_SCAN_WIDTH) &&
                        (rdr_prm.radar_del_az > -MAX_SCAN_WIDTH))
```

P5162A3.PDF [Page: 44 of 101]


```
                {
                        antenna_data_.new_azimuth_center = radar_az_center
                        radar_.RADAR.NEW_CENTER_IND = True
                        safire_.change_flag = true
                }
        }
                :
}

/* this method reads the parameters that the SSS has used to perform the current simulation and returns the
parameters as a Radar_Parameter_Set */
void Radar_Parameter_Set::get_Radar_Parameters()
{
                :
        radar_az_center = radar_.RADAR.AZ_CENTER;
                :
}
```

Modifications to SAFIRE

No modifications to existing SAFIRE code are anticipated to support a change in radar_az_center.

2.4.2.5 Radar_Parameter_Change.radar_el_center

This parameter represents the azimuthal center of the radar scan range.

Code for Radar_Parameter_Set

```
include safirefc.h, F_radar.h, F_sim.h


void Radar_Parameter_Set::update_parameters()
{
                :
        if radar_el_center ≠ Radar_.RADAR.EL_CENTER
        {
                // verify that the value is between -70 and 70 degrees
                if ((rdr_prm.radar_del_az <= MAX_SCAN_WIDTH) &&
                        (rdr_prm.radar_del_az > -MAX_SCAN_WIDTH))
                {
                        antenna_data_.new_elevation_center = radar_el_center
                        radar_.RADAR.NEW_CENTER_IND = True
                        safire_.change_flag = true
                }
        }
                :
}

/* this method reads the parameters that the SSS has used to perform the current simulation and returns the
parameters as a Radar_Parameter_Set */
void Radar_Parameter_Set::get_Radar_Parameters()
```

PS16249.PDF [Page: 45 of 101]


```
}
                    :
        radar_el_center = radar_.RADAR.EL_CENTER;
                    :
}
```

Modifications to SAFIRE
No modifications to existing SAFIRE code are anticipated to support a change in radar_el_center.

2.4.2.6 Radar_Parameter_Change.radar_del_az

This is the maximum of the azimuth angle (with respect to the center of the search azimuth) that the radar will scan. The angle should be in radian units. In order to input the new scan range it will have to be converted to degrees. Changes to this parameter can be handled with out modification of existing SAFIRE code.

Code for Radar_Parameter_Set

```
void Radar_Parameter_Set::update_parameters()
{
                    :
        if (radar_.RADAR.DEL_AZ != radar_del_az)
        {
                // verify that the value is between 0 and 70 degrees
                if ((rdr_prm.radar_del_az <= MAX_SCAN_WIDTH) &&
                        (rdr_prm.radar_del_az > 0))
                {
                        antenna_data_.new_azimuth_scan_range = radar_del_az * deg_to_rad.
                        radar_.RADAR.NEW_WIDTH_IND=TRUE;
                        safire_.change_flag=1;
                }
        }
                    :
}

/* this method reads the parameters that the SSS has used to perform the current simulation and returns the
parameters as a Radar_Parameter_Set */
void Radar_Parameter_Set::get_Radar_Parameters()
{
                    :
        radar_del_az = radar_.RADAR.DEL_AZ;
                    :
}
```

Modifications to SAFIRE
No modifications to existing SAFIRE code are anticipated to support a change in radar_del_az.

P516249.PDF [Page: 46 of 101]

2.4.2.7 Radar_Parameter_Change.radar_acq_ind

Not yet supported

2.4.2.8 Radar_Parameter_Change.TAGC

Not yet supported

2.4.2.9 Radar_Parameter_Change.radar_acq_bump

Not yet supported

2.4.2.10 Radar_Parameter_Change.radar_acq_az

Not Yet Supported

2.4.2.11 Radar_Parameter_Change.radar_acq_vel

Not Yet Supported

2.4.2.12 Radar_Parameter_Change.radar_acq_range

Not Yet Supported

2.4.2.13 Radar_Parameter_Change.radar_acq_scale

Not Yet Supported

2.4.2.14 Radar_Parameter_Change.radar_az

The radar_az is the equivelant of RADAR.AZ defined in the file radar.inc. This is the current azimuth of the radar beam. This is not a user definable parameter. The value is determined by SAFIRE as it simulates the APG-65 radar. This value is needed by the aircraft to display the current azimuthal position of the radar antenna.

This value should be updated to the value in SAFIRE and returned to the Aircraft at at an update rate of 30Hz..

```
/* this method reads the parameters that the SSS has used to perform the current simulation and returns the
parameters as a Radar_Parameter_Set */
void Radar_Parameter_Set::get_Radar_Parameters(){
                    :
        azimuth = radar_.RADAR.AZ;
                    :
}
```

P516249.PDF [Page: 47 of 101]

2.4.2.15 Radar_Parameter_Change.radar_el

The radar_el is the equivalent of the RADAR.EL defined in the file radar.inc. This is current elevation of the radar beam This is not a user definable parameter. The value is determined by SAFIRE as it simulates the APG-65 radar. This value should be updated and returned to the Aircraft at the same time as the Radar Status Report. This value is needed by the aircraft to display the current elevation angle of the radar antenna.

This value should be updated to the value in SAFIRE and returned to the Aircraft at at an update rate of 30Hz..

```
/* this method reads the parameters that the SSS has used to perform the current simulation and returns the
parameters as a Radar_Parameter_Set */
void Radar_Parameter_Set::get_Radar_Parameters()
{
                    :
        .elevation = radar_.RADAR.EL;
                    :
}
```

2.4.3 Radar Status Report

This section describes how the Radar_Status_Report will be generated and sent via the RTI. A Radar_Status_Report will be sent when a detection (or hit) is made by the radar. The report will consist of the detected range, closing velocity, azimuth and elevation for each detection (hit). An estimate of the target that was the source of the hit will also be included in the status report. The details of the attributes of a detection can be found in Table 3. There is an additional parameter in the report that indicated the type of report. In this case the type is a regular hit. This will be the only type of detections made in Range While Search (RWS) mode.

The Track While Scan (TWS) mode will generate track file report in addition to the regular detections. There are two types of track file reports that will be sent as a Radar_Status_Report. The first type of track file is a general track file. The general track files consist of all track files that do not have a rank of 0. The track file with a rank of 0 is reported as the launch and steer track file. In both cases an additional Radar_Status_Report attribute indicating the track quality will be sent as part of the Radar_Status_Report. The ranking algorithm is internal to SAFIRE and the status reporting object plays no part in ranking the track files.

P516249.PDF [Page: 48 of 101]

Table 3 Radar Detection Attributes

Type	Parameter	Description
int	hit_info_number_of_hits	Number of detections in this report
float	range	Range of a detected hit
float	velocity	Closing velocity of the hit
enum	hit_type	Type of report
float	azimuth	Azimuth of the detected hit
float	elevation	Elevation of the detected hit
int	track_quality	Quality of the track file
int	estimated_Target_ID	an estimate of the target aircraft for this detection

There are a total of fifteen types of reports that can possible be generated in SAFIRE. An enumeration of the type of possible reports can be found in Table 4. Since the radar modes have been restricted to RWS and TWS only three types of reports will be generated. These are the REGULAR, TWS_LS, TWS_R1-9 and TWS_GENERAL reports.

Table 4 Types of Radar_Status_Reports

Report Type	Description
REGULAR	Standard detection
NOISE	Noise detected at azimuth and elevation location
TWS_LS	Track While Scan Launch and Steer target
TWS_R1	Track While Scan target rank 1
TWS_R2	Track While Scan target rank 2
TWS_R3	Track While Scan target rank 3
TWS_R4	Track While Scan target rank 4
TWS_R5	Track While Scan target rank 5
TWS_R6	Track While Scan target rank 6
TWS_R7	Track While Scan target rank 7
TWS_R8	Track While Scan target rank 8
TWS_R9	Track While Scan target rank 9

TWS_GENERAL	Track While Scan track file
STT_REGULAR	Single Target Track target report
STT_ANGLE_ONLY_TRACK	Single Target Track with no range or velocity

Radar Object Code:

```
/* the method called from the GUI to handle the RTI communications */
void Radar_Object::Run()
{
        if (Do_Receive)
        {
                /* tick the RTI ambassador to deliver any interactions
                   The first parameter is the minimum time for the RTI Ambassador to take
                   The second  paramter is the maximum time for the RTI Ambassador
                During this time the RTI should deliver any updates to the targets, the ownship and radar
                parameters */
                My_RTIAmb.tick(0.1,1.0);

                Do_Recieve = FALSE;

        }
        else if (Do_Send and !(Do_Recieve))
        {
                :
                statusReport.get_status_report();
                if (statusReport.number_of_hits() != 0)
                {
                        sendRadarStatusReport(statusReport.make_publishable_status_report());
                }
                if (timeToUpdateTrackFile())
                {
                        statusReport.get_track_report()
                        if (statusReport.number_of_hits() != 0)
                        {
                                sendRadarStatusReport(statusReport.make_publishable_status_report());
                        }
                }
                :
        }
}
```

2.4.3.1 Class Definitions

Two classes are defined to handle the Radar Status Reports. Each hit is maintained by the Hit class. The Hit class will maintain the attributes of radar detections. These attributes are listed in Table 3. A Radar_Status_Report class is also defined to support the collecting and sending of these reports via an interaction in the RunTime Infrastructure (RTI).

P518249.PDF [Page: 50 of 101]

The Radar_Status_Report class will store the detections in a list of Hit objects. When a report is built from the detections/track files in SAFIRE a Hit object is created for each detection/track file and stored in the list. When the report is prepared to be sent as an interaction, it packages the hit data as an array of Hit_s data structures as well as an integer indicating the number of hits. The array has a maximum length of 10 hits. This maximum should be large enough to send all detections in RWS mode. In RWS mode detections usually do not occur in large numbers in a single coherent processing interval (cpi). In TWS mode SAFIRE has a limit of 10 track files so the limit on the number of hits won't interfere with sending all track files.

The definition of the Hit and Radar_Status_Report can be found in Appendix C. The method of extracting the hit information from SAFIRE and building a status report is listed here.

Code for Radar_Status_Report

```
/* This method generates the hit list and fills the data members with
   the relevant information from the Fortran common blocks, via the
   .inc files. */

Radar_Status_Report::get_status_report()
{

        hit_info_number_of_hits = detected_target_data_.numberof_detected_hits;

        for(i = 1; i < hit_info_number_of_hits; i++)
        {
                new Hit hit;

                hit.set_range(detected_target_data_.detected_target_range[i]);
                hit.set_vel(detected_target_data_.detected_target_velocity[i]);
                hit.set_hit_type(REGULAR);
                hit.set_azimuth(detected_target_data_.detected_target_azimuth[i]);
                hit.set_elevation(detected_target_data_.detected_target_elevation[i]);

                hit_info.push_back(hit);
        }
}
```

Modifications to SAFIRE

No modifications to support generation of the Radar_Status_Report.

2.4.4 Radar Pause Request and Radar Continue Request

These requests can be supported within the Radar Object. When a Radar_Pause_Request has been received the Radar_Object will wait until a Radar_Continue_Request is received before allowing execution to continue. D-SAFIRE will start in the paused state. This will allow it to receive the initial set of target and ownship data. The following is the implementation of the pause and continue requests.

Radar_Object Code

```
Boolean Radar_Pause = TRUE
Boolean Radar_Continue = FALSE

/* A method for processing this interaction will be called from the federate ambassador */
void Radar_Object::recieveInteraction(RTI::InteractionClassHandle theInteraction)
{
                :
        /* test for Radar Continue Request */
        if (theInteraction == Radar_Continue_Request_Interaction)
        {
                Radar_Continue = TRUE;
                Radar_Pause = FALSE;
        }

        /* test for Radar Pause Request */
        if (theInteraction == Radar_Pause_Request_Interaction)
        {
                Radar_Continue = FALSE;
                Radar_Pause = TRUE;
        }
                :
}

/* this method interacts with the RTI and SAFIRE */
void Radar_Object::run()
{
                :
        My_RTIAmb.tick();
        /* if a radar pause has been requests then wait for a
        radar continue request */
        while ((SAFIRE_pause == RTI::RTI_TRUE) && (SAFIRE_continue == RTI::RTI_FALSE))
        {
                if (SAFIRE_exit == RTI::RTI_TRUE) exit(0);

                /* wait for a radar continue interaction */
                My_RTIAmb.tick(0.1,1.0);
        }

                :
}
```

Modifications to SAFIRE Code

No modifications to support Radar_Pause_Requests or Radar_Continue_Requests.

2.4.5 Update Aircraft

This section describes how the Update_Aircraft interaction will be received and used to update the Ownship state in SAFIRE. The Update Aircraft interaction delivers the latest information on

P516249.PDF [Page: 52 of 101]

the state of the Ownship (the Aircraft the radar is associated with).

The RTI will deliver the Update_Aircraft interaction to our federate ambassador. Our federate ambassador will in turn deliver the interaction to the Radar_Object. The Radar_Object will examine the type of interaction, extract the data and hand the data to an Aircraft object to interpret. The Aircraft object will be responsible for interpreting the data and translating the data into values that can be inserted into SAFIRE.

The aircraft update attributes are defined in the Federation Object Model (FOM). The aircraft update consists of the current weapon configuration and a Time Space Position Inidcator defined for SAFIRE (SAFIRE_TSPI_s) These attributes are listed in Tables 5 - 9. The weapon configuration is an enumeration of the weapon configurations in the current SAFIRE. The time space position indicator consists of a position in altitude, latitude and longitude and a velocity and acceleration in North, East and Down directions.

Table 5 Aircraft Update Parameters

Type	Parameter	Description
SAFIRE_TSPI_s	TSPI	Time Space Position Indicator: defined for the simulation
enum wpn_cnf_s	wpn_cnf	Indicates the weapon selected

Table 6 Attrubutes of the SAFIRE_TSPI_s

Type	Parameter	Description
ALLVector	Position	Position in altitude, latitude and longitude
NEDVector	Velocity	Velocity in the north, east and down directions
NEDVector	Acceleration	Acceleration in the north, east and down directions

Table 7 Attributes of an ALLVector

Type	Parameter	Description
float	altitude	Distance above ground in metres
float	latitude	Latitude in radians
float	longitude	longitude in radians

PS16249.PDF [Page: 53 of 101]

Table 8 Attributes of a NEDVector

Type	Parameter	Description
float	north	North component
float	east	East component
float	down	Down component

Table 9 Enumeration of the Weapon Configurations

Weapon Configuration	Description
GUN	
AIM7F	
AIM7M	
AIM9L	
AIM9M	

Radar_Object Code

```
/* A method for processing this interaction will be called from the federate ambassador */
void Radar_Object::recieveInteraction(RTI::InteractionClassHandle theInteraction)
{
                    :
        /* test for Update Aircraft interaction */
        if (theInteraction == Udate_Aircraft_Interaction)
                Get the parameter handle
                check for expected parameter handle
                extract aircraft data  from the interaction
                set the ownship aircraft object attributes to the new values
                 :
}
```

2.4.5.1 Class Definitions

An Aircraft object was defined to encapsulate the data and processing necessary to receiving the ownship updates and inserting the values into SAFIRE. The Aircraft object will interpret updates received and store the data in an Aircraft_s structure. The Aircraft object will also perform any necessary conversion of the data into a form that can be put into SAFIRE.

The main component of the update is the time space position indication (TSPI). The TSPI data is held in a TSPI object. The TSPI object will maintain the TSPI data and provide the heading and speed based on the data. The TSPI object consists of a ALLvector object for the position data and

P516249.PDF [Page: 54 of 101]

two NEDVector objects for the velocity and acceleration. The ALLVector class contains the altitude, latitude and longitude values for a position. The NEDVector class contains the north, east and down components of a vector. The NEDVector class will be able to return the magnitude and heading of the vector. The complete definition of these classes can be found in Appendix C.

Modifications to SAFIRE

The modifications needed to support the Update_Aircraft interaction are the same as those defined for the Update_Target_Processing_List interaction.

2.4.6 Time Management

D-SAFIRE is intended to operate as part of a distributed real time simulation. As a result D-SAFIRE will have to receive target and ownship updates and generate hit updates at least as fast as real time. Running in real time means that we can depend on the clock in the machine to keep us in real time and have our time advance with the rest of the distribution. Thus we will not need to use any of the time management facilities available in the HLA. For more information on time management in the RTI read the RTI1.3NG Version 3.1 Programmer's Guide chapters 3 and 6 [2].

At start up we will have to note the start time of the simulation according to the clock in the machine. This could be done when the first complete set of target and ownship attributes are received. Advancement of time can be controlled by the clocks in the machines. The clocks should advance at the same rate and thus keep all simulation components synchronized.

D-SAFIRE must run at least as fast as real time in order for it to keep up with the rest of the simulation. If the amount of time simulated is greater than the real time D-SAFIRE can wait for the real time to catch up and then send the results. There is nothing that can be done if D-SAFIRE is slower than real time. If it does occur that D-SAFIRE is slower than real time The only noticeable effect would be that the antenna scan rate would be slower than expected.

The comparison of realtime and the simulated radar time is done comparing the real time it took to perform on coherent processing interval (CPI) and the amount of time simulated for the CPI. It was chosen to do the comparison on a per cpi basis because the simulated radar time only advances each CPI.

The RTI Programmer's Guide suggests that a call to tick (see Appendix A of RTI1.3NG Version 3.1 Programmer's Guide) should be used to slow an application participating in a HLA simulation. Through experimentation it was discovered that the minimum time for a call to tick is 0.01s. Thus delaying call to tick will only be made if the previous cpi was simulated 0.01s faster than real time.

The processing required to check the realtime execution of SAFIRE was encapsulated in a separate method of the Radar_Object. Thus the Run method of the Radar_Object needs to call this method to maintain the realtime execution.

P516249.PDF [Page: 55 of 101]

Radar_Object Code

```
void Radar_Object::run()
{
        :
        /* check real  time */
        check_real_time();
        :
} /* end run */

void Radar_Object::check_real_time()
{
        double delta;    // difference between the previous and current wall clock time
        float delay;     // the amount of time to delay the simulation

        /* if the simulation is ahead the wait for real time to catch up */
        clock_gettime(CLOCK_REALTIME,&currentTime);
        if (FirstTime == RTI::RTI_FALSE)
        {
          // determine the real time spent in the previous loop
            delta = (double) 1.0*(currentTime.tv_sec - prevTime.tv_sec) +
                (currentTime.tv_nsec - prevTime.tv_nsec) * 0.000000001;

                // add to the total real execution time
            duration += delta;

                // if simulated time was greater the real time the wait for
                // real time to catch up
                // the 0.01 added to delta is for the call overhead
            if (time_data_.radar_run_time > duration + 0.01)
            {
                    delay = time_data_.radar_run_time - duration - 0.01;
                    //printf("ahead of real time by %f seconds\n",delay + 0.01);

                    // tick the RTI Ambassador to cause a delay
                    //My_RTIAmb.tick(delay,delay+0.0001);
            }
            /*
            Received_Radar_Parameters.check_antenna_scan(
                    time_data_.radar_run_time - prevRunTime,
                    delta);
            */
        }
        else FirstTime = RTI::RTI_FALSE;

        // store current values for the next loop
        prevTime.tv_sec = currentTime.tv_sec;
        prevTime.tv_nsec = currentTime.tv_nsec;
        prevRunTime = time_data_.radar_run_time;
        prevEndTime = times_.END_TIME;
}
```

P516249.PDF [Page: 56 of 101]

2.4.7 Radar_Object Class Definition

This section summarizes the analysis presented in the previous sections in the form of a class definition for the Radar Object. The Radar_Object class was defined in order to control the interface between the RTI communication and the existing SAFIRE data structures.

The Radar_Object class consists of a single Aircraft, Target_List and Radar_Status_Report objects and two Radar_Parameters_Set objects. Using Two Radar_Parameters_Set objects allows the separation of the parameters received from the ACD and input into SAFIRE, and the parameters used by SAFIRE and sent back to the ACD. Each of these objects encapsulates the specifics of interfacing with the SAFIRE data structures. This includes converting between the data representation in SAFIRE and the data defined in the Federation Object Model [1].

There are several additional attributes and methods define in the Radar_Object. These additional methods are used to perform RTI communication. The Radar_Object class definition is complete as of the writing of this document and can be found in Appendix D. A specification of the code for the Radar_Object:: Run method is listed in Appendix E.

2.5 D-SAFIRE Federate Ambassador

The federate ambassador for D-SAFIRE will have to support the call backs that will be generated as a result of the activities of the Radar_Object. The federate ambassador class has been defined by the RTI. The methods the we need to support are determined by the activity of the Radar_Object. The activities of the Radar_Object can be categorized as object management and declaration management.

Declaration management call backs ensure that there is another federation interested in the interactions that the Radar_Object can produce. Ideally, the Radar_Object should not publish interactions or objects that no one in interested in. This will reduce the amount of wasted band width.

The object management functions are for receiving objects and interactions from the RTI. In our case, we are only interested in receiving interactions. Thus we need only support the receiveInteraction method.

Below are the federate ambassador functions that we need to support. For more information of the federate ambassador please refer to the RTI1.3NG Version 3.1 Programmer's Guide [2].

Declaration Management

```
virtual void turnInteractionsOn (
  RTI::InteractionClassHandle theHandle) // supplied C1
throw (
  RTI: InteractionClassNotPublished,
  RTI::FederateInternalError) { }

virtual void turnInteractionsOff (
```

P516249.PDF [Page: 57 of 101]


```
  RTI::InteractionClassHandle theHandle) // supplied C1
throw (
  RTI::InteractionClassNotPublished,
  RTI::FederateInternalError) { }
```

Object Management

```
virtual void receiveInteraction (
      RTI::InteractionClassHandle      theInteraction, // supplied C1
  const RTI::ParameterHandleValuePairSet& theParameters, // supplied C4
  const RTI::FedTime&                 theTime,      // supplied C4
  const char                          *theTag,      // supplied C4
      RTI::EventRetractionHandle      theHandle)    // supplied C1
throw (
  RTI::InteractionClassNotKnown,
  RTI::InteractionParameterNotKnown,
  RTI::InvalidFederationTime,
  RTI::FederateInternalError) { }


virtual void receiveInteraction (
      RTI::InteractionClassHandle      theInteraction, // supplied C1
  const RTI::ParameterHandleValuePairSet& theParameters, // supplied C4
  const char                          *theTag)      // supplied C4
throw (
  RTI::InteractionClassNotKnown,
  RTI::InteractionParameterNotKnown,
  RTI::FederateInternalError) { }
```

P516249.PDF [Page: 58 of 101]

3. Testing

Testing for the D-SAFIRE program should demonstrate that the program still provides a high fidelity radar simulation, can receive updates and publish results as part of a distributed simulation and no additional software defects have been added.

The existing SAFIRE program has been established as a benchmark for producing a realistic radar simulation. A detailed test ensuring that D-SAFIRE can provide a high fidelity radar simulation is difficult. It is assumed that the existing SAFIRE code, which has not been modified is correct. The method we have chosen to verify that D-SAFIRE operates as well as SAFIRE is to generate output of the radar hits generated by SAFIRE. The same output can then be generated for D-SAFIRE and compared to the original for a similar physical situation. The comparison of output will be used to verify the high fidelity operation of D-SAFIRE.

The distributed communication can be verified by providing a test stub program that will take the place of all other parts of the distributed simulation. The test stub program will send updates for the targets, ownship and radar parameters. The test stub will also receive the output generated by D-SAFIRE.

An additional test program will operate as an observer and verify the communication between D-SAFIRE and the Test Stub. The observer will be able to listen for all interactions between the test stub and D-SAFIRE or the D-SAFIRE parent. The observer will be able to display the data received in the interactions.

3.1 Testing Interface

User interfaces for testing should convey information in a human readable form. The testing programs will be able to write text messages to the interface. The messages will indicate what events have occurred in the test.

The user interface for the test stub will consist of a console. The test stub program will display what information it is sending to the D-SAFIRE program.

The observer program will display what information is being sent by both the D-SAFIRE program and the test stub program. The observer program provides an independent verification on the interactions between the test stub and the D-SAFIRE modules. There are two user interfaces for the Observer program. One version will write text messages to the console. A second version has a Graphical User Interface (GUI) which is used to display text messages.

3.2 Test Observer

The Observer using the console as the user interface will be limited as to the interactions it can listen for. This limitation is a result of the limited user interface. The user will be able to specify

P516249.PDF [Page: 59 of 101]

which interactions it is interested in by using command line options when starting the observer program. The observer will display the data contained in the interactions of interest. In order to save on network traffic the interactions not being displayed will not be subscribed to. The observer defaults to listening for updates to the target data. The following defines the command line options for the Observer program:

Table 10 Observer Command Line Parameters

Option	Description
-t	Listen for target updates (default)
-o	Listen for ownship updates
-p	Listen for parameter updates
-s	Listen for radar status reports

This version is implemented on the Solaris and Windows98/NT machines

The Observer using the GUI will listen for all communication in the federation and display the latest data received in interactions. The GUI will maintain a brief history of the last 100 updates received for each interaction. The size of the history maintained is set at compile time. This version of the Observer is designed for use on Windows98/NT.

The structure of both the Observer program with the GUI and without the GUI matches that used for the Radar_Object class in D-SAFIRE.Much of the implementation can be copied from D-SAFIRE. There are some small differences between the Observer class used in the GUI and non-GUI versions. The Observer class with the GUI has some attributes defined which facilitate communication with the user interface components. The non-GUI version has some attributes defined to selectively subscribe to interactions based on the command line parameter.

The Federate Ambassador for both Observer programs will have to support receiving interactions and the call backs from the RTI for federation information. As mentioned before the federate ambassador class has been defined by the RTI. The following are the methods that we will need to support for the observer federate and they are listed below. For more information on the federate ambassador please refer to the RTI1.3NG Version 3.1 Programmer's Guide [2].

Declaration Management

```
virtual void turnInteractionsOn (
  RTI::InteractionClassHandle theHandle) // supplied C1
throw (
  RTI::InteractionClassNotPublished,
  RTI::FederateInternalError) { }

virtual void turnInteractionsOff (
  RTI::InteractionClassHandle theHandle) // supplied C1
```

P516249.PDF [Page: 60 of 101]


```
throw (
  RTI::InteractionClassNotPublished,
  RTI::FederateInternalError) { }

Object Management
virtual void receiveInteraction (
      RTI::InteractionClassHandle      theInteraction, // supplied C1
  const RTI::ParameterHandleValuePairSet& theParameters, // supplied C4
  const RTI::FedTime&                  theTime,      // supplied C4
  const char                          *theTag,       // supplied C4
      RTI::EventRetractionHandle       theHandle)    // supplied C1
throw (
  RTI::InteractionClassNotKnown,
  RTI::InteractionParameterNotKnown,
  RTI::InvalidFederationTime,
  RTI::FederateInternalError) { }

virtual void receiveInteraction (
      RTI::InteractionClassHandle      theInteraction, // supplied C1
  const RTI::ParameterHandleValuePairSet& theParameters, // supplied C4
  const char                          *theTag)       // supplied C4
throw (
  RTI::InteractionClassNotKnown,
  RTI::InteractionParameterNotKnown,
  RTI::FederateInternalError) { }
```

3.3 Test Stub design

The test stub must provide the interactions that all other parts of the distributed simulation would be expected to provide. This means the test stub must publish updates of the ownship and targets. The update rate has been defined in the Federation Object Model [1] as 30Hz. The updates will have values for all the parameters defined in the Federation Object Model. The test flights that will be used are defined in section 3.5. The data for the test flights was generated by running the scenarios in SAFIRE and recording the Ownship and Target attributes in files. The test stub must also receive the Radar Status Reports and parameters sent by D-SAFIRE

The user interface will indicate what the test stub has published and received. This output needs only state if it has sent any updates or received any interactions but may also output the values sent or received.

The update method of the Aircraft and Target_List objects is overloaded to allow the Test Stub to pass a FILE pointer. The FILE pointer indicates where to get the next set of data for the ownship/targets. This method will then read the next set of ownship/target data from the file.

As with the Observer the structure of the Test Stub program is the same as that of the Radar_Object class in D-SAFIRE.

The Test_Stub will also need a federate ambassador. The federate ambassador for the test stub

needs to support the same object management and declaration management as D-SAFIRE. The following are the methods that we need to support in the federate ambassador for the Test_Stub.

Declaration Management

```
virtual void turnInteractionsOn (
  RTI::InteractionClassHandle theHandle) // supplied C1
throw (
  RTI::InteractionClassNotPublished,
  RTI::FederateInternalError) { }

virtual void turnInteractionsOff (
  RTI::InteractionClassHandle theHandle) // supplied C1
throw (
  RTI::InteractionClassNotPublished,
  RTI::FederateInternalError) { }
```

Object Management

```
virtual void receiveInteraction (
      RTI::InteractionClassHandle      theInteraction, // supplied C1
  const RTI::ParameterHandleValuePairSet& theParameters,  // supplied C4
  const RTI::FedTime&                 theTime,       // supplied C4
  const char                          *theTag,       // supplied C4
      RTI::EventRetractionHandle       theHandle)     // supplied C1
throw (
  RTI::InteractionClassNotKnown,
  RTI::InteractionParameterNotKnown,
  RTI::InvalidFederationTime,
  RTI::FederateInternalError) { }

virtual void receiveInteraction (
      RTI::InteractionClassHandle      theInteraction, // supplied C1
  const RTI::ParameterHandleValuePairSet& theParameters,  // supplied C4
  const char                          *theTag)       // supplied C4
throw (
  RTI::InteractionClassNotKnown,
  RTI::InteractionParameterNotKnown,
  RTI::FederateInternalError) { }
```

3.4 Test Plan

The first stage in testing will be to define a baseline of performance of the existing program. A Radar Output File (ROF) was generated while running the test flight defined in section 3.5. The ROF contains the values that would make up the Radar Status Report. The ROF generated from the test flights will act as a baseline for the operation of SAFIRE. We can use this baseline to compare with the output of D-SAFIRE. When an equivalent flight is run on the ACD if the ROF is within expected tolerance (To Be Defined) we can be confident that the simulation software is operating as well as it was before any modifications were made.

P516249.PDF [Page: 62 of 101]

The next stage of testing will be to ensure that the D-SAFIRE parent can successfully create and kill the D-SAFIRE process. The start and exit of the D-SAFIRE program can be noted visually by the operator.

It is suggested that the RTIexec operate on a different computer. This will eliminate an additional step in testing the RTIexec running on the same machine and then on a different machine. The functionality of the RTIexec can be demonstrated by the chat program prototype, documented in the D-SAFIRE Software Prototypes [4] document.

The next stages test the D-SAFIRE parent's ability to receive the Create_Radar_Object, Destroy_radar_Object and Terminate_Simulation interactions. This test will require some test stub functionality in order to generate these interactions. The observer program should be working as well in order to verify the correct publication of the interactions. The effect of these interactions can be verified visually by the operator but a file of the interactions received by the D-SAFIRE parent should also be made to compare with the interactions sent.

The next stage should be to test the Radar_Pause_Request and the Radar_Continue_Request interactions. These interactions should cause D-SAFIRE to stop and start. This behaviour can be noted by the operator and a log file of the interactions received should be made by the Radar_Object. This test stage will require the additional functionality built into the test stub and the observer to verify publication of the interactions.

At this point we can start testing some of the more complex interactions. These involve publishing and receiving parameters via the RTI and interfacing with the variables in the SSS component. The first stage in this testing will be to have D-SAFIRE publish a Radar_Status_Report and a set of Radar_Parameters. These interactions will be received by the observer program. The received interactions will be written to the user interface and to a log file. The observer could also generate a Radar Output File (ROF) matching the format of the baseline ROF generated.

Modifications to the radar parameters should be tested. The radar parameters should be reflected when D-SAFIRE publishes the Radar_Parameters. The request to change to the parameters and the resulting change in the reflected parameters can be noted in the output of the observer.

At this point D-SAFIRE should still be using it's own profile generator as a source of updates to the ownship and targets. Using the same test flights, the ROF from this test should be the same as that for the baseline. The two ROFs should be compared to ensure correct operation at this point.

The next stage will be to supply the updates to the ownship and targets through the RTI. The test stub will be required to publish the updates at 30Hz. The test stub used a Kinematic Output File (KOF), generated by saving the flight dynamics generated in SAFIRE, as updates for the ownship and targets. D-SAFIRE will publish the Radar_Status_Reports as in the previous testing stage. The ROF generated can again be compared to the baseline version to ensure correct operation.

P516249.PDF [Page: 63 of 101]

3.5 Test Flight Scenarios

These test flights will be used to generate a baseline performance of SAFIRE. The baseline will establish that the current SAFIRE works and provide something to compare the operation of D-SAFIRE with.

The baseline will be generated by writing to a file the values that would go into the radar status report as defined in the Federate Object Model. After modifications to SAFIRE. The same output file can be generated with the same scenarios. The two files can be compared to verify that SAFIRE is operating as it was before modifications.

We have devised two test scenarios to use for generating a baseline for SAFIRE, the T-Flight and the head on flight.

3.5.1 T-Flight Parameters

Description: In this flight the two aircraft (one Ownship and one Target) will fly in orthogonal trajectories with the Target aircraft flying in front of the Ownship (aircraft with the radar). After the initial conditions as specified in **Figure 5** both aircraft will fly in straight lines at constant speed. There is no change in elevation for either aircraft. The simulation should run for 60 seconds.

P516249.PDF [Page: 64 of 101]

Figure 5 T-Flight initial Conditions

Table 11 T-Flight Parameters

Ownship	Target
initial Latitude = 10.0° Longitude = 35.00° Speed = 300.00 knots Altitude = 4000.00 feet Heading = 10.0°	initial Range = 20.00 km Azimuth = -45.00° Velocity = 800.00 knots Altitude = 4000.00 feet Heading = 280.00°
End of Segment 1 Duration = 60.00 seconds Final Speed = 300.00 knots Final Altitude = 4000.00 feet Final Heading = 10.0°	End of Segment 1 Duration = 60.00 seconds Final Speed = 800.00 knots Final Altitude = 4000.00 feet Final Heading = 280.00°

P516249.PDF [Page: 65 of 101]

3.5.2 Head-On Flight Parameters

Description: Two aircraft (one Ownship, one Target) approach head on. The two aircraft start at a distance of 14 km, Ownship and Target ground speed are initially 300 knots, both aircraft at an elevation of 4000 feet. The initial Ownship heading is 0.0° and the initial Target heading is 180°. The Ownship will fly at a constant ground speed of 300 knots, while the Target accelerates steadily from 300 knots to 400 knots ground speed for 8 seconds (Segment 1, t = 0s to t = 8s). Over the next 8 seconds (Segment 2, t = 8s to t = 16s). the Ownship will steadily accelerate from a ground speed of 300 knots to 450 knots, diving at a constant rate from 4000 feet to 3500 feet and turning at a constant angular velocity from a heading of 0° to 45°. Over Segment 2 the Target aircraft will accelerate at a constant rate from a ground speed of 400 knots to 550 knots, while climbing at a constant rate from 4000 feet to 4500 feet and turning with a constant angular velocity from a heading of 180° to 225°. Over the final 30 seconds (Segment 3, t = 16s to t = 46s) the Ownship and Target aircraft will continue at a constant speed, heading and elevation. The total duration for the simulation should be 46 seconds. The flight characteristics are illustrated in Figures ? and **Figure 6**, and summarized in Table 12

P516249.PDF [Page: 66 of 101]

Table 12 Head-On Flight Parameters

Ownship	Target
initial Latitude = 10.0° Longitude = 0.0° Speed = 300.0 knots Altitude = 4000.0 feet Heading = 0.0°	initial Range = 14.0 km Azimuth = 0.0° Velocity = 300.0 knots Altitude = 4000.0 feet Heading = 180.0
Segment 1 Duration = 8.0 sec Final Speed = 300.0 Final Altitude = 4000.0 Final Heading = 0.0°	Segment 1 Duration = 8.0 sec Final Speed = 400.0 knots Final Altitude = 4000.0 feet Final Heading = 180.0°
Segment 2 Duration = 8.0 sec Final Speed = 450.0 knots Final Altitude = 3500.0 feet Final Heading = 45.0°	Segment 2 Duration = 8.0 sec Final Speed = 550.0 knots Final Altitude = 4500.0 feet Final Heading = 225.0°
Segment 3 Duration = 30.0 sec Final Speed = 400.0 knots Final Altitude = 3500.0 feet Final Heading = 45.0°	Segment 3 Duration = 30.0 sec Final Speed = 550.0 knots Final Altitude = 4500.0 feet Final Heading = 225.0°

P516249.PDF [Page: 67 of 101]

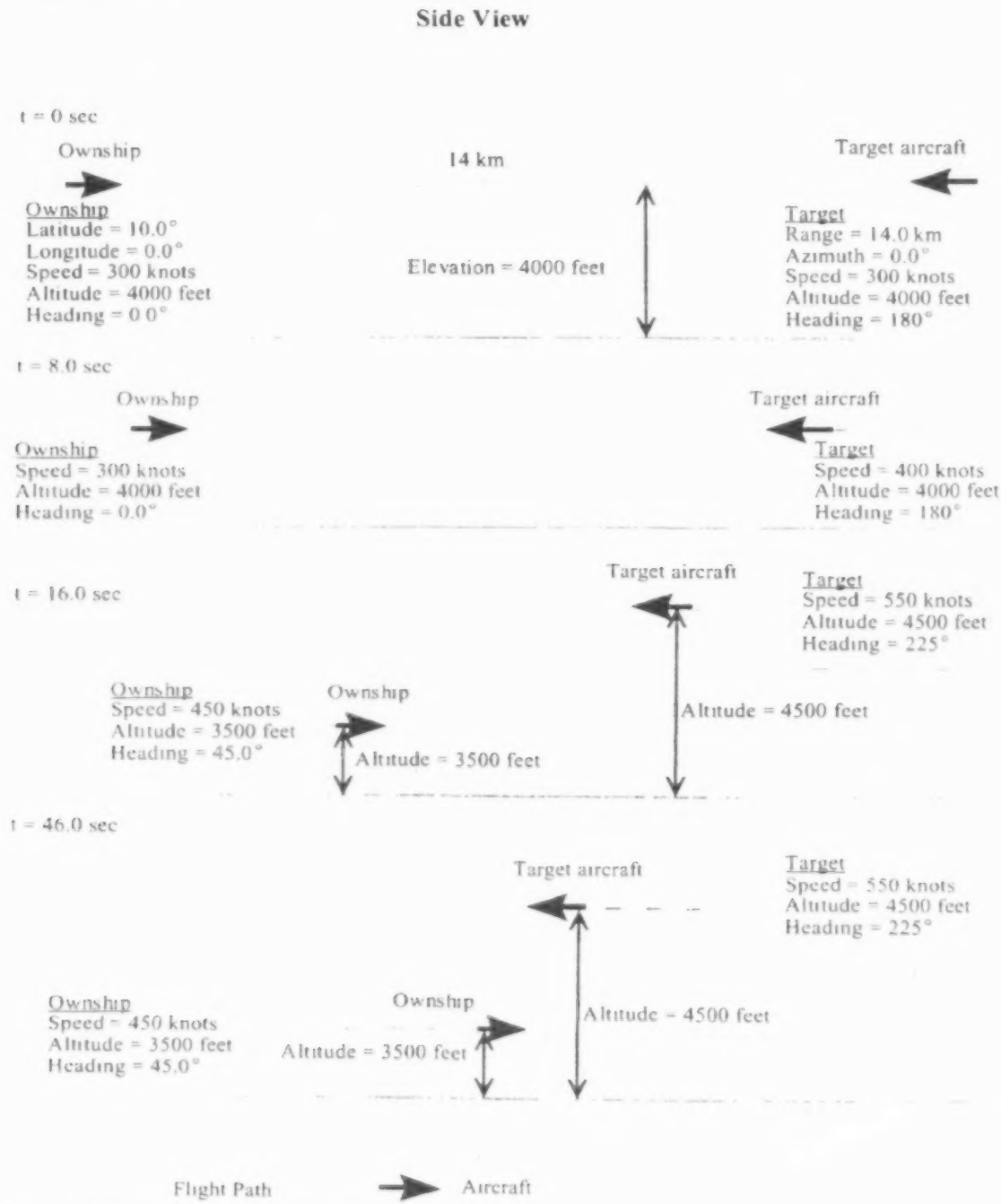

Figure 6 Head-On Flight Side View

P516249.PDF [Page: 68 of 101]

References

[1] D-SAFIRE Federation Object Model
Vantage Point International
November, 2000

[2] High-Level Architecture Run-Time Infrastructure RTI 1.3-Next Generation Programmer's Guide Version 3.1
Department of Defense, Defense Modelling and Simulation Office
July, 1999

[3] Distributed MAWS Engagement Program Summary Report
Canadian Marconi Company
Document Number 1000-1186
March 1999

[4] D-SAFIRE Software Protoypes
Vantage Point International
November, 2000

P516249.PDF [Page: 69 of 101]

Appendix A - Class Definition of D_SAFIRE_Parent

```
class D_SAFIRE_Parent
{
public:
        /* turns off the interaction  indicated by theHandle */
        void interactionOff(RTI::InteractionClassHandle theHandle);

        /* turns the interaction indicated by theHandle on */
        void interactionOn(RTI::InteractionClassHandle theHandle);

        /* processes all incomming interactions */
        void receiveInteraction(RTI::InteractionClassHandle theInteraction,
                const RTI::ParameterHandleValuePairSet& theParameters,
                const char *theTag);

        /* used for testing only */
//      Parent_Federate_Ambassador *get_fed_amb();

/* does the main processing of listening for requests to create and destroy the
  D-SAFIRE process as well as terminate simulation */
        void run();

/* set up the RTI communications */
        void init();

/* constructor
        Responsible for and initializations */
        D_SAFIRE_Parent();

/* used for testing only */
        D_SAFIRE_Parent(FILE *logfile);

/* destructor
        responsible for ending communications with the RTI */
        ~D_SAFIRE_Parent();

private:
        void ResignAndDestroy();

        /* use the RTI ambassador to get a parameter handle for a specific
        interaction class */
        RTI::ParameterHandle makeParameterHandle(
                const char *parameterName,
                RTI::InteractionClassHandle theInteraction);

        /* use the RTI ambassador to get an interaction class handle for a specific
        interaction class */
        RTI::InteractionClassHandle makeInteractionClassHandle(
                const char *className);

        /* methods for sending specific interactions */
```

```
        void sendDestroyRadarObject();
        void sendCreateRadarObject();
        void sendRadarContinueRequest();
        void sendRadarPauseRequest();
        /* send a Radar Parameter interactions */
        void sendRadarParameters(struct Radar_Parameters_s theRadarParameters_s);

        /* methods that help to terminate communications with the RTI
        unsubscribe/unpublish to all interactions */
        void unpublishInteractions();
        void unsubscribeInteractions();

        /* method to unpublish/unsubscribe to specific interaction classes */
        void unpublish(RTI::InteractionClassHandle theInteraction);
        void unsubscribe(RTI::InteractionClassHandle theInteraction);

        /* indicate which interactions I can produce */
        void publishInteractions();

        /* subscribe to all relevant interactions */
        void subscribeInteractions();

        /* publush a specific interaction class */
        void publish(RTI::InteractionClassHandle theInteraction);

        /* subscribe to a specific interaction class */
        void subscribe(
                RTI::InteractionClassHandle theInteraction,
                RTI::Boolean active);

        /* get the handles all interactions I will be interasted in */
        void getInteractionHandles();

        /* name of the Federation Execution Details (FED) file */
        char *const FED_File_Name;

        /* name of the Federation */
        char *const FederationName;

/* the RTI ambassador used to communicate with the rest of the distribution*/
        RTI::RTIambassador My_RTIAmb;

/* my federate Ambassador, used by the RTI to communicate with me */
//      Parent_Federate_Ambassador My_FedAmb;

/* a flag indicateing the state of the D-SAFIRE process */
        int DEAD;

/* A method that starts a new D-SAFIRE process */
        void start();

/* the process ID of D-SAFIRE */
```

```
        pid_t SAFIRE;
//      int SAFIRE;

        RTI::AttributeHandleSet *federateTypeHandleSet;

        RTI::Boolean SAFIRE_exists;
        RTI::Boolean SAFIRE_destroyed;

        /* flags that control the operation of SAFIRE */
        RTI::Boolean SAFIRE_exit;                   // kill the safire process
        RTI::Boolean SAFIRE_execute;    // create a new safire process
        RTI::Boolean SAFIRE_terminate;  // end of simulation
        RTI::Boolean SAFIRE_pause;
        RTI::Boolean SAFIRE_continue;

        /* interaction class handles */
        RTI::InteractionClassHandle TerminateSimulationHandle;
        RTI::InteractionClassHandle CreateRadarObjectHandle;
        RTI::InteractionClassHandle DestroyRadarObjectHandle;
        RTI::InteractionClassHandle RadarPauseRequestHandle;
        RTI::InteractionClassHandle RadarContinueRequestHandle;
        RTI::InteractionClassHandle RadarParametersHandle;

        /* parameter handles */
        RTI::ParameterHandle SAFIRE_terminateHandle;
        RTI::ParameterHandle SAFIRE_executeHandle;
        RTI::ParameterHandle SAFIRE_exitHandle;
        RTI::ParameterHandle SAFIRE_pauseHandle;
        RTI::ParameterHandle SAFIRE_continueHandle;
        RTI::ParameterHandle rdr_prmHandle;

        /* interaction flags which indicate the interactions I can send */
        RTI::Boolean TerminateSimulationInteraction;
        RTI::Boolean CreateRadarObjectInteraction;
        RTI::Boolean DestroyRadarObjectInteraction;
        RTI::Boolean RadarPauseRequestInteraction;
        RTI::Boolean RadarContinueRequestInteraction;
        RTI::Boolean RadarParametersInteraction;

/* the current parameters used by SAFIRE */
        Radar_Parameters_Set theRadarParameters;

/* used for reset timer */
        time_t current_time;
        time_t start_time;

/* the time when D-SAFIRE needs to be reset */
        time_t reset_time;

/* the longest uninterupted execution time for D-SAFIRE */
        time_t max_sim_time;
};
```

P516249.PDF [Page: 72 of 101]

Appendix B - Mode Changes in SAFIRE

This appendix illustrated how we might support additional modes in D-SAFIRE. As mentioned in the requirements section, the restriction to one mode of operation might be expanded in future. This is how mode changes might be handled.

```
void Radar_Object::update_parameters(struct Radar_Parameter_Change Radar_Parameters)
{
                :
/* check if the radar mode has changed */
        if (Radar_Parameter_Change.radar_mode_ind <> radar_.RADAR.MODE_IND)
        {
                Case: Radar_Parameter_Change.radar_mode_ind of
                VS_MODE:        new_mode(VS)
                RWS_HPRF_MODE:
                        If radar_.RADAR.MODE_IND = RWS_MPRF_MODE
                                New_prf_ind = true
                                New_prf_mode = HPRF_RWS
                        else
                                New_mode(RWS)
                        Break;
                RWS_MPRF_MODE:
                        If Radar_.RADAR.MODE_IND = RWS_HPRF_MODE
                                New_prf_ind = true
                                New_prf_mode = MPRF_RWS
                        else
                                New_mode(RWS)
                        Break;
                TWS_MODE:
                        not supported
                STT_MODE:
                        not supported
        }
                :
}
```

P516240.PDF [Page: 73 of 101]

Appendix C - Additional Class and Structure Definitions

```
/* the maximum number of targets currently allowable in SAFIRE */
#define MAX_NUMTARG 10

enum rcs
{
        CONSTANT = 0,
        SWERLING_MODEL_1,
        SWERLING_MODEL_2
};


/* structure for holding the electronic counter measure information */
struct ecm_cnf_s
{
        /* jammer status */
        RTI::Boolean jammer_on;
};


/* class for processing the electronic counter measure information */
class ecm_cnf
{
public:
        /* constructors */
        /* default constructor */
        ecm_cnf();

        /* constructor with a ecm_cnf_s structure */
        ecm_cnf(struct ecm_cnf_s new_ecm_cnf);

        /* constructor with an array of bytes */
        ecm_cnf(char *new_ecm_cnf);

        /* constructor with a boolean */
        ecm_cnf(RTI::Boolean jammer_state);

        /* default destructor */
        virtual ~ecm_cnf() {};

        /* modification methods */
        /* set a new value to the jammer status */
        void set_jammer(RTI::Boolean jammer);

        /* set a new electronic counter measure configuration structure */
        void set_jammer(struct ecm_cnf_s new_jammer);

        /* set a new electronic counter measure configuration structure */
        void set_jammer(char * new_jammer);

        /* access methods */
```

P516249.PDF [Page: 74 of 101]


```
        /* return the status of the jammer */
        RTI::Boolean get_jammer();

        /* return the electronic counter measure configuration structure */
        struct ecm_cnf_s get_jammer_configuration();

        /* print out the contents of the ecm_cnf_s structure */
        void out();

private:

        /* electronic counter measure configuration */
        struct ecm_cnf_s jammer_config;
};


/* structure to hold the electromagnetic characteristics */
struct em_cnf_s
{
        float target_rho_magnitude;

        /* target radar cross section */
        enum rcs_model target_rcs_model;
};


/* class to handle the electro-magnetic data */
class em_cnf
{
public:
        /* constructors */
        /* default constructor */
        em_cnf();

        /* construct a new em_cnf object given the em_cnf structure */
        em_cnf(struct em_cnf_s new_em_cnf);

        /* construct a new em_cnf object given an array of bytes */
        em_cnf(char *new_em_cnf);

        /* default destructor */
        virtual ~em_cnf() {};

        /* modification methods */
        /* assign a new value to the target_rcs_model */
        void set_target_rcs_model(enum rcs_model model);

        /* assign a new set of values to the whole em_cnf structure */
        void set(struct em_cnf_s new_em_cnf);

        /* assign a new set of values to the whole em_cnf structure */
        void set(char * new_em_cnf);
```

```
        /* assign a new value to the target rho magnitude */
        void set_target_rho_magnitude(float magnitude);

        /* access methods */
        /* return the RCS model */
        enum rcs_model get_target_rcs_model();

        /* return the rho magnitude */
        float get_target_rho_magnitude();

        /* return the whole electromagnetic configuration */
        struct em_cnf_s get();

        /* print out the contents of this_em_cnf */
        void out();

private:
        /* structure to hold the electro magnetic configuration data */
        struct em_cnf_s this_em_cnf;
};


/* a structure to hold the data of the vector
this is used so that the data can be transmitted
over the RTI */
struct NEDVector_s
{
        float north;
        float east;
        float down;
};


struct ALLVector_s
{
        float latitude;
        float longitude;
        float altitude;
};


/* the class of NEDVector
this should be used to manipulate the NEDVector structure and the
values it containes */
class NEDVector
{
public:
        /* constructor methods */
        /* default constructor */
        NEDVector();

        /* the initial values of the components are passed as three
```

```
        floating point numbers */
        NEDVector(float an_north, float an_east, float an_down);

        /* a vector structure is passed which has the initial values */
        NEDVector(struct NEDVector_s new_vector);

        /* read the initial values from an array of bytes */
        NEDVector(char *theVector);

        /* modifier methods */
        void set_east(float new_east);          // new value for east
        void set_north(float new_north);        // new value for north
        void set_down(float new_down);          // new value for down
        void set(struct NEDVector_s new_vector);  // new values for north, east, down
        void set(char * new_vector);            // new values for north, east, down

        /* accessor methods */
        float get_down();                       // return value of down
        float get_east();                       // return value of east
        float get_north();                      // return value of north
        struct NEDVector_s get(); // return the structure with all values

        /* returns the heading for the velocity */
        float get_heading();

    /* returns the magnitude of teh vector */
    float get_magnitude();

        /* print values */
        void out();

        /* default destructor */
        virtual ~NEDVector() { };

private:
        /* vector structure to hold the data */
        struct NEDVector_s thisVector;
};


class ALLVector
{
public:
        /* constructor methods */
        /* default constructor */
        ALLVector();

        /* the initial values of the components are passed as three
        floating point numbers */
        ALLVector(float an_altitude, float an_latitude, float an_longitude);

        /* a vector structure is passed which has the initial values */
```

```
        ALLVector(struct ALLVector_s new_vector);

        /* read the initial values from an arralatitude of blatitudetes */
        ALLVector(char *theVector);

        /* modifier methods */
        void set_latitude(double new_latitude);        // new value for latitude
        void set_altitude(double new_altitude);        // new value for altitude
        void set_longitude(double new_longitude);  // new value for longitude
        void set(struct ALLVector_s new_vector);   // new values for altitude, latitude, longitude
        void set(char * new_vector);           // new values for altitude, latitude, longitude

        /* accessor methods */
        double get_longitude();                        // return value of longitude
        double get_latitude();                         // return value of latitude
        double get_altitude();                         // return value of altitude
        struct ALLVector_s get(); // return the structure with all values

        /* print values */
        void out();

        /* default destructor */
        virtual ~ALLVector();
private:
        /* vector structure to hold the data */
        struct ALLVector_s thisVector;
};


/* a time-space position indication structure */
struct SAFIRE_TSPI_s
{
        struct ALLVector_s position;
        struct NEDVector_s velocity;
        struct NEDVector_s acceleration;
};


class TSPI
{
public:
        /* consructor */
        /* default */
        TSPI();

        /* build from an array of bytes */
        TSPI(char *aTSPI);

        /* build for a TSPI structure */
        TSPI(struct SAFIRE_TSPI_s new_TSPI);

        /* destructor */
```

P516240.PDF [Page: 78 of 101]


```
        virtual ~TSPI() { };

        /* access methods */
        struct NEDVector_s get_acceleration();
        struct NEDVector_s get_velocity();
        struct ALLVector_s get_position();
        float get_heading();   // returns the heading for the TSPI
        float get_speed();
        float get_altitude();
        struct SAFIRE_TSPI_s get();

        /* modification methods */
        void set_acceleration(struct NEDVector_s anacceleration);
        void set_velocity(struct NEDVector_s avelocity);
        void set_position(struct ALLVector_s aposition);
        void set(struct SAFIRE_TSPI_s new_TSPI);
        void set(char * new_TSPI);
        void update(float timestep);

        /* convert the TSPI representation to a representation required for SAFRIE */
        struct safire_state_s convert_to_safire();

        /* print out the contents */
        void out();

private:
        /* vector objects to process and hold the vectors */
        NEDVector acceleration;
        NEDVector velocity;
        ALLVector position;

        /* structure to hold the TSPI data */
        struct SAFIRE_TSPI_s thisTSPI;
};


/* a structure for all information regarding a target */
struct Target_Status_s
{
        int ID;
        int visible;
        struct ecm_cnf_s a_ecm_cnf;
        struct em_cnf_s a_em_cnf;
        float range;
        float radial_velocity,
        float radial_acceleration;
        float azimuth;
        float elevation;
};


class Target_Status
```

```
{
public:

        /* constructor */
        /* default constructor */
        Target_Status();

        /* constructor with a target status structure */
        Target_Status(struct Target_Status_s status);

        /* constructor with an array of bytes */
        Target_Status(char *),

        /* default destructor */
        virtual ~Target_Status() {};

        /* access methods */
        /* return the Identification number */
        int get_ID();

        /* return the electronic counter measure configuration structure */
        struct ecm_cnf_s get_ecm_cnf();

        /* return the electromagnetic configuration structure */
        struct em_cnf_s get_em_cnf();

        /* get visibility */
        int get_visibility();

        /* return the range */
        float get_range();

        /* return the radial velocity */
        float get_radial_velocity();

        /* return the radial acceleration */
        float get_radial_acceleration();

        /* return the azimuth */
        float get_azimuth();

        /* return the elevation */
        float get_elevation();

        /* return the Target Status structure */
        struct Target_Status_s get_status();

        /* printf out the contents of the Target Status structure */
        void out();

        /* modification methods */
        /* set the identification number */
```

P516249.PDF [Page: 80 of 101]


```
        void set_ID(int new_ID);

        /* set visibility */
        void set_visibility(int new_visibility);

        /* set the electronic counter measure configuration */
        void set_ecm_cnf(struct ecm_cnf_s new_ecn_cnf);

        /* set the electromagnetic configuration */
        void set_em_cnf(struct em_cnf_s new_em_cnf),

        /* set the range */
        void set_range(float new_range);

        /* set the radial velocity */
        void set_radial_velocity(float new_radial_velocity);

        /* set the radial acceleration */
        void set_radial_acceleration(float new_radial_acceleration);

        /* set the azimuth angle */
        void set_azimuth(float new_azimuth);

        /* set the elevation angle */
        void set_elevation(float new_elevation);

        /* set a new target status */
        void set_status(struct Target_Status_s new_status);

        /* set a new target status */
        void set_status(char * new_status);

        /* calculate an updated status based on a time step */
        void update(float timestep, float ownshipDistance, float elevationDistance);

private:
        /* the ecm_cnf object responisbel for maintaining the ecm_cnf data */
        ecm_cnf My_ecm_cnf;

        /* the em_cnf object responsibe for maintaining the em_cnf data */
        em_cnf My_em_cnf;

        /* the target status structure */
        struct Target_Status_s status;
};


struct Target_List_s
{
    /* uniquely identified each instantiation of this structure */
    int sequence_number;
```

```
        /* number of targets */
        int number_of_targets;

        /* the list of targets */
        struct Target_Status_s Target[HLA_NUMTARG];
};

class Target_List
{
public:

        /* constructors */
        /* default constructor */
        Target_List();

        /* constructor given a Target List to build from */
        Target_List(struct Target_List_s new_List);

        /* constructor given an array of bytes to build from */
        Target_List(char *newList);

        /* default destructor */
        virtual ~Target_List() {};


        /* modification methods */
        /* add a target to the list */
        RTI::Boolean add(Target_Status aTarget);

        /* add a target to the list */
        RTI::Boolean add(struct Target_Status_s aTarget);

        /* set new values the target list */
        void set(const char * new_status);

        /* insert the target stats into SAFIRE */
        void update(float ownship_altitude);

        /* genetate updated range and velocity based on the current range, velocity
          and acceleration, the time step and the speed of the Ownship Aircraft */
        void update(float timestep, float ownshipSpeed, float elevationDistance);


        /* access methods */
        /* returns the target status of the ith target */
        struct Target_Status_s get_target(int i);

        /* returns the list of targets */
        struct Target_List_s *get_target_list();

        /* get the number of targets */
```

P516249.PDF [Page: 82 of 101]


```
        int get_number_of_targets();

        /* print the contents of the target list */
        void out();

        /* write out a comma delimited line of target data */
        void write_out();

private:
        /* file to read the updates from */
        FILE *targetfp;

        /* the list of target the data for all targets */
        struct Target_List_s targets;

        /* the list of target objects */
        Target_Status theTarget[10];
};


Class Scale
{
Public:
        float: rdelta
        float: vdelta
};


enum radar_modes
{
        VS_MODE = 0,        /* velocity search not supported */
        RWS_HPRF_MODE,      /* Range while search, high pulse repetition mode */
        RWS_MPRF_MODE,      /* Range while search, medium pulse repetition mode */
        RWS_INT_MODE,       /* not supported */
        TWS_MODE,           /* track while scan, not supported */
        STT_MODE,           /* single target track mode, not supported */
        NO_MODE,            /* no mode selected (default value) */
};


/* structure for holding the radar parameters */
struct Radar_Parameters_s
{
        /* radar mode */
        enum radar_modes radar_mode_ind;

        /* number of elevation bars */
        int radar_numbar;

        /* azimuth center of the antenna scan */
        float radar_az_center;
```

```
        /* elevation of the antenna scan */
        float radar_el_center;

        /* maximum azimuth angle wrt the scan centre */
        float radar_del_az;

        /* the following 7 parameters are not currently supported */
        RTI::Boolean radar_acq_ind;
        float TAGC;
        RTI::Boolean acq_bump;
        float acq_az;
        float acq_el;
        float acq_vel;
        float acq_range;
        struct Scale acq_scale;

        /* the current azimuth angle of the antenna */
        float azimuth;
        /* the current elevation angel of the antenna */
        float elevation;
};

Class Radar_Parameter_Sets
{
public:

        /* constructors */
        /* default constructor */
        Radar_Parameters_Set();

        /* construct from a Radar Parameter set structure */
        Radar_Parameters_Set(struct Radar_Parameters_s theParamters);

        /* construct from an array of bytes */
        Radar_Parameters_Set(char *theParameters);

        /* default destructor */
        virtual ~Radar_Parameters_Set() {};

        /* modification methods */
        /* set the radar parameters */
        void set_parameters(struct Radar_Parameters_s new_rdr_prm);

        /* set the radar parameters */
        void set_parameters(char * new_rdr_prm);

        /* build from the parameters in safire */
        void build_parameters();

        /* check that the antenna is scanning in real time */
        void check_antenna_scan(float radar_time, double real_time);
```

```
        /* access methods */
        /* extract the radar parameters */
        struct Radar_Parameters_s get_parameters();

        /* return the radar mode */
        enum radar_modes radar_mode();

        /* use the current radar parameters to update the values in safire */
        void update_parameters();

        /* print out the contents of the radar parameters */
        void out();

        /* print out the contents of the radar parameters to fit the header */
        void demo_out();

        /* print out the contents of the radar parameters */
        void headder_out();
private:
        /* the structure controlled by this object */
        struct Radar_Parameters_s rdr_prm;
};


struct Hit_s
{
        float range;
        float vel;
        enum hit_info_hit_type hit_type;
        float azimuth;
        float elevation;
        int track_quality;
        int estimatedTargetID;
};


enum hit_info_hit_type
{
        REGULAR = 0,
        NOISE_TYPE,
        TWS_LS,
        TWS_R1,
        TWS_R2,
        TWS_R3,
        TWS_R4,
        TWS_R5,
        TWS_R6,
        TWS_R7,
        TWS_R8,
        TWS_R9,
        TWS_GENERAL,
        STT_REGULAR,
```

```
        STT_ANGLE_ONLY_TRACK
};

class Hit
{
public:
        /* default constructor */
        Hit();

        /* construct from a Hit_s structure */
        Hit(struct Hit_s aHit);

        /* construct from an array of bytes */
        Hit(char *);

        /* default destructor */
        virtual ~Hit();

        /* modification methods */
        void set_range(float new_range);
        void set_vel(float new_vel);
        void set_hit_type(enum hit_info_hit_type new_hit_type);
        void set_azimuth(float new_azimuth);
        void set_elevation(float new_elevation);
        void set_track_quality(int quality);
        void set_estimatedTargetID(int ID);
        void set(struct Hit_s new_Hit);
        void set(char *aHit);

        /* access methods */
        float get_range();
        float get_vel();
        enum hit_info_hit_type get_hit_type();
        float get_azimuth();
        float get_elevation();
        int get_track_quality();
        int get_estimatedTargetID();
        struct Hit_s get();

        /* generate a printable version of the contents of the object */
        void out();
private:
        /* the structure to hold the data */
        struct Hit_s theHit;
};

struct rdr_sts_s
{
        int hit_info_number_of_hits;
        struct Hit_s theHit[MAX_HITS];
};
```

P516249.PDF [Page: 86 of 101]


```
class Radar_Status_Report
{
public:
        /* default constructor */
        Radar_Status_Report();

        /* consruct from a array of byte that resemble the structure rdr_sts_s */
        Radar_Status_Report(char * aStatus);

        /* default destructor */
        virtual ~Radar_Status_Report();

        /* get set values from and array of byte that resemble the structure rdr_sts_s */
        void set(char * aStatus);

        int number_of_hits();

        /* return the specified hit */
        Hit get_hit(int hit_number);

        /* build the status report from the data in SAFIRE */
        void make_status_report();

        /* build the status report from a Hit object */
        void make_status_report(Hit aHit);

        /* builds a report of the track files */
        void make_track_report();

        /* match hits to targets */
        void match_hits_to_targets(Target_List_s * targets);

        /* rate the match between the hit and the target */
        float rate(Hit_s * aHit, Target_Status_s *aTarget);

        /* make a status report that can be published as an array of bytes */
        struct rdr_sts_s make_publishable_status_report();

        /* the size (in bytes) of the status report */
        size_t size();

        /* print the contents of the object */
        void out();

        /* print the contents of the object to a file */
        void out(FILE *fp, float time);

        /* print the contents of the object in a table to the stdout
        Format: Counter,Range,Velocity,Hit_type,Azimuth,elevation */
        void demo_out();
```

P618249.PDF [Page: 87 of 101]


```
        /* print fout the headder for the output of demo_out() */
        void headder_out();

private:
        /* the number of hits */
        int hit_info_number_of_hits;

        /* the list of hits */
        RWGSlist(Hit) hit_info;
};


enum weapon
{
        GUN = 1,
        AIM7F = 2,
        AIM7M = 3,
        AIM9L = 4,
        AIM9M = 5
};


struct Aircraft_s
{
        int sequence_number;
        struct SAFIRE_TSPI_s my_TSPI;
        enum weapon wpn_cnf;
};


class Aircraft
{
public:
        /* constructor */
        Aircraft();

        /* construct the aircraft object from an array of bytes */
        Aircraft(char *);

        /* construct the aircraft from an aircraft structure */
        Aircraft(struct Aircraft_s new_Aircraft);

        /* destructor */
        virtual ~Aircraft() {};

        /* modification methods */
        /* set the TSPI structure */
        void set_TSPI(struct SAFIRE_TSPI_s new_TSPI);

        /* set the weapon */
        void set_weapon(enum weapon new_weapon);
```

P516249.PDF (Page: 88 of 101)


```
        /* define a new set of values for the aircraft object */
        void set(struct Aircraft_s new_Aircraft);

        /* define a new set of values for the aircraft object */
        void set(char * new_Aircraft);

        /* access mehtods */
        /* return the TSPI structure */
        struct SAFIRE_TSPI_s get_TSPI();

        /* return the weapon configuration */
        enum weapon get_weapon();

        /* return the Aircraft structure */
        struct Aircraft_s get();

        /* return the aircraft altitude */
        float get_altitude();

        /* update values in the SSS component of SAFIRE */
        void update();

        /* update the values in the TSPI based on the time step */
        void update(float timestep);

        /* print the contents of the object */
        void out();

private:
        TSPI currentTSPI;

        /* current state of the aircraft */
        Aircraft_s currentAircraft;
};
```

Appendix D - Class Definition of the Radar_Object

```
Class Radar_Object
{
public:
        /* this method handles all the RTI communication during normal
        execution */
        void run();

        /* this method sets up all the RTI communication */
        void init();

        /* default constructor
          initializes any flags and variables */
        Radar_Object();

        /* default destructor
          shuts down the RTI communication */
        ~Radar_Object();

        /* these methods are called by the Federate ambassador to deliver
          any communication from the RTI */

        /* method the turn on interactions for a specific interaction class */
        void InteractionsOn(RTI::InteractionClassHandle theHandle);

        /* turns off interactions for a specific interactions class */
        void InteractionsOff(RTI::InteractionClassHandle theHandle);

        /* processes incomming interactions */
        void receiveInteraction(RTI::InteractionClassHandle theInteraction,
                const RTI::ParameterHandleValuePairSet &theParameters,
                const char *theTag);

private:
        void ResignAndDestroy();

        /* objects to handle specific data structures */

        /* object to handle the radar status reports */
        Radar_Status_Report statusReport;

        /* structures and object to handle radar parameters */
        struct Radar_Parameters_s rdr_prm;
        Radar_Parameters_Set Current_Radar_Parameters;
        Radar_Parameters_Set Received_Radar_Parameters;


        /* structures and object to handle aircraft updates */
        struct Aircraft_s aircraftState;
        Aircraft My_Aircraft;
```

```
/* structure and object to handle target updates */
struct Target_List_s trg_prl;
Target_List theTargetList;

/* intitialization helper methods */

/* this method waits for the initial set of aircraft, target and parameter
values */
void get_initial_data();

/* this method wait for all the published interactions to be subscribed to */
void waitInteractionsOn();

/* get interection class handles and parameter handles for
RTI publication and subscription */
void getInteractionHandles();

/* makes a object class handle */
RTI::ObjectClassHandle makeObjectClassHandle(const char *ClassName);

/* makes the attribute handle set */
RTI::AttributeHandleSet *makeAttributeHandleSet(RTI::ObjectClassHandle theClass);

/* make attribute handles */
RTI::AttributeHandle makeAttributeHandle(const char *AttributeName,
                    RTI::ObjectClassHandle theClass);

/* publish all interactions I can generate */
void publishInteractions();

/* publish the object and attributes that I will maintain */
void publishObjectsandAttributes();

/* subscribe to all interactions I can process */
void subscribeInteractions();

/* use the RTI ambassador to get a parameter handle for a specific
interaction class */
RTI::ParameterHandle makeParameterHandle(
        const char *parameterName,
        RTI::InteractionClassHandle theInteraction);

/* use the RTI ambassador to get an interaction class handle for a specific
interaction class */
RTI::InteractionClassHandle makeInteractionClassHandle(
        const char *className);

/* pubiush a specific interaction class */
void publish(RTI::InteractionClassHandle theInteraction);

/* subscribe to a specific interaction class */
void subscribe(
```

```
                RTI::InteractionClassHandle theInteraction,
                RTI::Boolean active);

        /* destructor helper methods */

        /* unpublish all interactions I can publish */
        void unpublishInteractions();

        /* unsubscribe to all interactions I can process */
        void unsubscribeInteractions();

        /* method to unpublish/unsubscribe to specific interaction classes */
        void unpublish(RTI::InteractionClassHandle theInteraction);
        void unsubscribe(RTI::InteractionClassHandle theInteraction);

        /* RTI communication methods */

        /* sends a RadarContinueRequest as confirmation to receiving
        a RadarContinueRequest */
        void sendRadarContinueRequest();

        /* sends RadarPauseRequest as confirmation to receiving a
        RadarPauseRequest */
        void sendRadarPauseRequest();

        /* send a CreateRadarObject interaction as confirmation of
          creation of this object */
        void sendCreateRadarObject();

        /* send the Radar parameters used for th ssimulation */
        void sendRadarParameters(struct Radar_Parameters_s theRadarParameters);

        /* send a report of any detections by the radar */
        void sendRadarStatusReport(struct rdr_sts_s aStatusReport);

        /* update the radar parameters attribute */
        void updateRadarParameters(struct Radar_Parameter_s theRadarParameters);


        /* timer methods */
        /* this method keeps the program from running faster that real time */
        void check_real_time();

        /* this method will indicate if it is time to send a parameter update
      and update the GUI */
        RTI::Boolean timeToUpdateParameters();

        /* attributes */

        /* the RTI ambassador for D-SAFIRE federate */
        RTI::RTIambassador My_RTIAmb;
```

```
/* name of the Federation Execution Details (FED) file */
char *const FED_File_Name;

/* name of the Federation */
char *const FederationName;

/* control flags to indicate what occured during the last call to the
   RTI Ambassador */

/* a flag to indicate that a RadarContinueRequest has to
be send as confirmation of receiving a RadarContinueRequest */
RTI::Boolean Confirm_continue;

/* a flag to indicate that a RadarPauseRequest has to
be send as confirmation of receiving a RadarPauseRequest */
RTI::Boolean Confirm_pause;

/* a flag to indicate that an Aircraft update was received */
RTI::Boolean AircraftUpdateReceived;

/* a flag to indicate that a Target update was received */
RTI::Boolean TargetUpdateReceived;

/* a flag to indicate that a parameter update was received */
RTI::Boolean ParametersReceived;

/* a used to toggle processing updates or sending results */
RTI::Boolean Do_Receive;

/* interaction generation attributes */

/* this is generated for each interaction
   it will contain the parameter handle and the data associated
   with the parameter */
RTI::ParameterHandleValuePairSet *aParameterHandleValuePairSet;

/* Factory objects */
/* this object creates a new ParameterHandleValuePairSet */
RTI::ParameterSetFactory aParameterSetFactory;

/* This object creates a new AttributeHandleSet,
   which is used when registering an Object instance */
RTI::AttributeHandleSetFactory APG_65AttributeHandleSetFactory;

/* This bject is used to generate attribute handle value pair sets */
RTI::AttributeSetFactory APG_65AttributeSetFactory;

/* flags that control the operation of SAFIRE */
RTI::Boolean SAFIRE_execute;
RTI::Boolean SAFIRE_exit;
RTI::Boolean SAFIRE_pause;                  // pause execution
RTI::Boolean SAFIRE_continue;   // continue execution
```

```
/* note: SAFIRE_continue = !SAFIRE_Pause */

/* Object class handles */
RTI::ObjectClassHandle RadarClassHandle;
RTI::ObjectHandle APG_65Handle;

RTI::AttributeHandle APG_65ParameterHandle;
RTI::AttributeHandle APG_65StatusHandle;
RTI::AttributeHandle APG_65TargetProcessingListHandle;

/* interaction class handles
   These are determined by the RTI Ambassador
   A unique handle is generated for each interaction.
   The interactions are defined int the .FED file */
RTI::InteractionClassHandle CreateRadarObjectHandle;
RTI::InteractionClassHandle DestroyRadarObjectHandle;
RTI::InteractionClassHandle RadarPauseRequestHandle;
RTI::InteractionClassHandle RadarContinueRequestHandle;
RTI::InteractionClassHandle UpdateAircraftHandle;
RTI::InteractionClassHandle UpdateTargetProcessingListHandle;
RTI::InteractionClassHandle RadarParametersHandle;
RTI::InteractionClassHandle RadarStatusReportHandle;

/* interaction flags
   These flags indicate if someone is interested in the interactions
   This federate can generate */
RTI::Boolean CreateRadarObjectInteraction;
RTI::Boolean RadarPauseRequestInteraction;
RTI::Boolean RadarContinueRequestInteraction;
RTI::Boolean RadarParametersInteraction;
RTI::Boolean RadarStatusReportInteraction;

/* parameter handles
   These are generated by the RTI AMbassador
   A unique handle is generated for each parameter
   The parameters are defined in the .FED file */
RTI::ParameterHandle SAFIRE_executeHandle;
RTI::ParameterHandle SAFIRE_exitHandle;
RTI::ParameterHandle SAFIRE_pauseHandle;
RTI::ParameterHandle SAFIRE_continueHandle;
RTI::ParameterHandle AircraftHandle;
RTI::ParameterHandle trg_prlHandle;
RTI::ParameterHandle rdr_prmHandle;
RTI::ParameterHandle rdr_stsHandle;

/* Timmer attributes */
/* current wall clock time and time the simulation was started */
struct timespec currentTime;   // current wall clock time
struct timespec prevTime;      // previous wall clock time
/* note: struct timespec
      {
      long tv_sec;  // time in seconds
```

```
        long tv_nsec;  // time in nanoseconds
    };
*/

double duration;              // total duration of the simulation; sumin of delta
float prevRunTime;            // previous simulated radar time
float prevEndTime;            // previous real time in fortran code
RTI::Boolean FirstTime;       // used to indicate first time through
RTI::Boolean FirstParameters;

double timeOfPreviousParameterUpdate;    // last time sent updated parameters and updated the GUI

int number_of_target_updates;
int number_of_target_deliveries;
int prev_number_of_target_updates;
float prevUpdateTime;

int cpi;
int cpi_since_last_tick;
int max_cpi_to_wait;
int targetupdatesduringtick;

/* used to capture the target and ownship profiles */
void capture_target_profile();
void capture_ownship_profile();
FILE *targetfp;
FILE *ownshipfp;

/* used to capture parameter profiles */
void capture_parameters();
FILE *parameterfp;

/* used to capture the detections */
void capture_detections();
FILE *detectionfp;
FILE *statusfp;
};
```

Appendix E - Radar_Object::Run()

This appendix presents the code for the method Radar_Object::Run().

```
/* the method called from the GUI to handle the RTI communications */
void Radar_Object::run()
{
//      printf("Running\n");

        if (SAFIRE_exit == RTI::RTI_TRUE)
        {
            ResignAndDestroy();
            exit(0);
        }

        if (FirstTime == RTI::RTI_TRUE)
        {
            get_initial_data();
            printf("Received initial data\n");
        }

        /* confirm receiving a radar pause request */
        if (Confirm_pause == RTI::RTI_TRUE)
        {
                sendRadarPauseRequest();
                Confirm_pause = RTI::RTI_FALSE;
        }

        /* if a radar pause has been requests then wait for a
        radar continue request */
        while ((SAFIRE_pause == RTI::RTI_TRUE) && (SAFIRE_continue == RTI::RTI_FALSE))
        {
                if (SAFIRE_exit == RTI::RTI_TRUE) exit(0);

                /* wait for a radar continue interaction */
                My_RTIAmb.tick(0.1,1.0);
        }

        /* confirm receiving a radar continue request */
        if (Confirm_continue == RTI::RTI_TRUE)
        {
                sendRadarContinueRequest();
                Confirm_continue = RTI::RTI_FALSE;
        }

        /* either receive updates or send results */
        if (Do_Receive == RTI::RTI_TRUE)
        {
                Do_Receive = RTI::RTI_FALSE;
                /* tick the RTI to deliver any interactions */
                //while(My_RTIAmb.tick(0.01,0.1) == RTI::RTI_TRUE);
```

PS1024$.PDF [Page: 96 of 101]


```
// optimized tick algorithm
if (cpi_since_last_tick >= max_cpi_to_wait)
{
    My_RTIAmb.tick(0.004,0.008);
        targetupdatesduringtick = number_of_target_updates - prev_number_of_target_updates;
        if (targetupdatesduringtick < 1)
        {
            max_cpi_to_wait++;
        }
        else if (targetupdatesduringtick >= 2)
        {
                if (max_cpi_to_wait > 0)
                        max_cpi_to_wait -= targetupdatesduringtick;
        }
}

if (AircraftUpdateReceived == RTI::RTI_TRUE)
{
        AircraftUpdateReceived = RTI::RTI_FALSE;
        /* insert the updated aircraft data into safire */
        //printf("new Aircraft data:\n");
        //My_Aircraft.out();
        My_Aircraft.update();

        //capture_ownship_profile();

}
else if (FirstTime == RTI::RTI_FALSE)
{
        My_Aircraft.update(time_data_.radar_run_time - prevRunTime);
}

if (TargetUpdateReceived == RTI::RTI_TRUE)
{
        TargetUpdateReceived = RTI::RTI_FALSE;
        number_of_target_deliveries++;
        /* insert the updated target data into safire */
        //printf("new Target data\n");
        //theTargetList.out();
        theTargetList.update(My_Aircraft.get_altitude());

        /* output some statistics */
        if (FirstTime == RTI::RTI_FALSE)
        {
                printf("\r%i, freq=%6.2f, new= %i, %6.2f, %6.2f, %6.2f, %6.2f, cpi=%i, %i\n",
                number_of_target_updates,
                ((float) number_of_target_updates)/duration,
                targetupdatesduringtick,
                ((float) number_of_target_updates)/((float) cpi),
                ((float) number_of_target_updates)/((float) number_of_target_deliveries),
                duration,
```

P516249.PDF [Page: 97 of 101]


```
                    time_data_.radar_run_time,cpi, cpi_since_last_tick);

                    //Current_Radar_Parameters.headder_out();
                    //Current_Radar_Parameters.demo_out();
            }
            prevUpdateTime = duration;
            prev_number_of_target_updates = number_of_target_updates;

            //capture_target_profile();

            cpi_since_last_tick = 0;
    }
    else if (FirstTime == RTI::RTI_FALSE)
    {
     /* interpolate the status of each target */
            SAFIRE_TSPI_s aTSPI = My_Aircraft.get_TSPI();
            NEDVector_s * velocity = &(aTSPI.velocity);
            float NEspeed = sqrt(velocity->north * velocity->north +
                    velocity->east * velocity->east);
            float timestep = time_data_.radar_run_time - prevRunTime;
            theTargetList.update(timestep,NEspeed,velocity->down);
            //printf("interpolated Target data\n");
            //theTargetList.out();
    }

    /* get the initial parameters that were read from the sim_init.dat file */
    if ((FirstParameters == RTI::RTI_TRUE) && (ParametersReceived == RTI::RTI_FALSE))
    {
            FirstParameters = RTI::RTI_FALSE;
            printf("getting initial parameters\n");
            Received_Radar_Parameters.build_parameters();
            //capture_parameters();
    }

    /* overwrite any changes from the GUI */
    Received_Radar_Parameters.update_parameters();
    //printf("received parameters\n");
    if (ParametersReceived == RTI::RTI_TRUE)
    {
            ParametersReceived = RTI::RTI_FALSE;
            //Received_Radar_Parameters.headder_out();
            //Received_Radar_Parameters.demo_out();

            //capture_parameters();
    }

    /* this method keeps the program from running faster that real time
      also sets the flag FirstTime to RTI::RTI_FALSE */
    check_real_time();
    cpi_since_last_tick++;
    cpi++;
```

P516249.PDF [Page: 98 of 101]


```
        }
        else
        {
                Do_Receive = RTI::RTI_TRUE;
                /* send the latest status report and parameters if there is
                an interested federate
                also send the track files if we are in TWS mode*/

                if (RadarParametersInteraction == RTI::RTI_TRUE)
                {
                        if (timeToUpdateParameters() == RTI::RTI_TRUE)
                        {
                                Current_Radar_Parameters.build_parameters();
                                sendRadarParameters(Current_Radar_Parameters.get_parameters());
                                //printf("Parameters used: \n");
                                //Current_Radar_Parameters.headder_out();
                                //Current_Radar_Parameters.demo_out();
                                //printf("RadarParametersHandle = %d\n",
                                //    RadarParametersHandle);

                                if ((Received_Radar_Parameters.radar_mode() == radar_modes(TWS_MODE)) &&
                                        (RadarStatusReportInteraction == RTI::RTI_TRUE))
                                {
                                        statusReport.make_track_report();
                                        if (statusReport.number_of_hits() != 0)
                                        {
                                                statusReport.match_hits_to_targets(theTargetList.get_target_list());
                                                sendRadarStatusReport(statusReport.make_publishable_status_report());
                                                //statusReport.out(statusfp,time_data_.radar_run_time);
                                                //printf("track Report:\n");
                                                //statusReport.demo_out();
                                        }
                                }
                        }
                }

                if (RadarStatusReportInteraction == RTI::RTI_TRUE)
                {
                        statusReport.make_status_report();
                        if (statusReport.number_of_hits() != 0)
                        {
                                statusReport.match_hits_to_targets(theTargetList.get_target_list());
                                //printf("Status Report:\n");
                                //statusReport.demo_out();
                                //statusReport.out(statusfp,time_data_.radar_run_time);
                                sendRadarStatusReport(statusReport.make_publishable_status_report());
                        }
                }
                /* used to capture the target and ownship profiles for testing */
                //capture_detections();
        }
}
```

PS16245.PDF [Page: 99 of 101]

List of Symbols, Abbreviations, Acronyms and Initialisms

HLA	High Level Architecture
SAFIRE	Simulator for Advanced FIghter Radar ECCM development
D-SAFIRE	Distributed - Simulator for Advanced FIghter Radar ECCM development
RWS	Range While Scan
TWS	Track While Scan
ACD	Air Crewstation Demonstrator
ROF	Radar Output File
FOM	Federation Object Model
RTI	RunTime Infrastructure
CPI	Coherent Processing Interval
TAGC	Total Automatic Gain Control
GUI	Graphical User Interface
DMSO	Defence Modelling and Simulation Office
IEEE	Institute of Electrical and Electronics Engineers

P516249.PDF [Page: 100 of 101]

UNCLASSIFIED
SECURITY CLASSIFICATION OF FORM
(highest classification of Title, Abstract, Keywords)

DOCUMENT CONTROL DATA

(Security classification of title, body of abstract and indexing annotation must be entered when the overall document is classified)

1 ORIGINATOR (the name and address of the organization preparing the document. Organizations for whom the document was prepared, e.g. Establishment sponsoring a contractor's report, or tasking agency, are entered in section 8.)

Vantage Point International Inc.
1704 Carling Avenue
Ottawa, ON Canada K2A 1C7

2 SECURITY CLASSIFICATION (overall security classification of the document including special warning terms if applicable)

UNCLASSIFIED

3 TITLE (the complete document title as indicated on the title page. Its classification should be indicated by the appropriate abbreviation (S,C or U) in parentheses after the title.)

D-SAFIRE Software Design Document (U)

4 AUTHORS (Last name, first name, middle initial)

Williams, Craig; Seymour, Micheal; Horn, Todd

5 DATE OF PUBLICATION (month and year of publication of document)

July 2001

6a NO. OF PAGES (total containing information. Include Annexes, Appendices, etc.)

94

6b NO. OF REFS (total cited in document)

4

7 DESCRIPTIVE NOTES (the category of the document, e.g. technical report, technical note or memorandum. If appropriate, enter the type of report, e.g. interim, progress, summary, annual or final. Give the inclusive dates when a specific reporting period is covered.)

Contractor Report

8 SPONSORING ACTIVITY (the name of the department project office or laboratory sponsoring the research and development. Include the address.)

Aerospace Radar and Navigation
Defence Research Establishment Ottawa, Shirley's Bay
3701 Carling Avenue, Ottawa, Ontario, K1A 0Z4

9a PROJECT OR GRANT NO. (if appropriate, the applicable research and development project or grant number under which the document was written. Please specify whether project or grant)

13dn01

9b CONTRACT NO. (if appropriate, the applicable number under which the document was written)

W7714-0-0366

10a ORIGINATOR'S DOCUMENT NUMBER (the official document number by which the document is identified by the originating activity. This number must be unique to this document.)

DRDO CR 2001-066

10b OTHER DOCUMENT NOS. (Any other numbers which may be assigned this document either by the originator or by the sponsor)

11 DOCUMENT AVAILABILITY (any limitations on further dissemination of the document, other than those imposed by security classification)

(X) Unlimited distribution
( ) Distribution limited to defence departments and defence contractors; further distribution only as approved
( ) Distribution limited to defence departments and Canadian defence contractors; further distribution only as approved
( ) Distribution limited to government departments and agencies; further distribution only as approved
( ) Distribution limited to defence departments; further distribution only as approved
( ) Other (please specify)

12 DOCUMENT ANNOUNCEMENT (any limitation to the bibliographic announcement of this document. This will normally correspond to the Document Availability (11). However, where further distribution (beyond the audience specified in 11) is possible, a wider announcement audience may be selected.)

P516249.PDF [Page: 101 of 101]

13 ABSTRACT (a brief and factual summary of the document It may also appear elsewhere in the body of the document itself It is t desirable that the abstract of classified documents be unclassified Each paragraph of the abstract shall begin with an indication of security classification of the information in the paragraph (unless the document itself is unclassified) represented as (S), (C), or (U) It is not necessary to include here abstracts in both official languages unless the text is bilingual)

This document describes the software design for D-SAFIRE as part of contract W7714-0-0366. The purpose of the project is to migrate the existing high fidelity radar simulation program, SAFIRE, to an HLA compliant program, D-SAFIRE, which can then operate as part of a distributed simulation. D-SAFIRE will then be used as part of an Air Crewstation Demonstrator for the CF-18.

This document begins with an examination of the requirements for the project. This section breaks down the requirements into four areas. These areas are the requirements for the User Interface, RTI communications, Modifications to SAFIRE and Testing. As part of the requirements three conceptual models are presented as an aid to understanding the components of the required system and what is outside the scope of the system. The Aircraft conceptual model describes how the aircraft components and attributes are related. The conceptual model of the RTI communication illustrates the communication between the distributed components. The conceptual model of SAFIRE and D-SAFIRE shows the delineation between the software components of programs. The final part of the requireme section is a description of some Use Cases. These Use Cases are meant to illustrate the various circumstances and events that will occur for D-SAFIRE.

The software design section analyses each of the requirements and provides solutions to meet them. The solutions are presented as either belonging in the new RTI communication modules or as modifications to the existing SAFIRE co or both. Each of the interactions in the Federation Object Model are analysed and solutions proposed.

14 KEYWORDS, DESCRIPTORS or IDENTIFIERS (technically meaningful terms or short phrases that characterize a document and could helpful in cataloguing the document They should be selected so that no security classification is required Identifiers such as equip model designation, trade name, military project code name, geographic location may also be included If possible keywords should be from a published thesaurus e g Thesaurus of Engineering and Scientific Terms (TEST) and that thesaurus-identified If it is not pos select indexing terms which are Unclassified, the classification of each should be indicated as with the title)

pulse Doppler radar
distributed simulation
realtime simulation
High Level Architecture
software design
APG-65

#516249

CA011672